涉密论文 □　公开论文 ☑

浙 江 大 學

本科生毕业论文

题目　知识驱动的物联网嵌入式固件自动化根因分析方法研究

姓名与学号　张乔 3200102817

指导教师　纪守领

年级与专业　2020级 计算机科学与技术

所在学院　计算机科学与技术学院

递交日期　递交日期

浙江大学本科生毕业论文（设计）承诺书

1. 本人郑重地承诺所呈交的毕业论文（设计），是在指导教师的指导下严格按照学校和学院有关规定完成的。

2. 本人在毕业论文（设计）中除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得**浙江大学**或其他教育机构的学位或证书而使用过的材料。

3. 与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

4. 本人承诺在毕业论文（设计）工作过程中没有伪造数据等行为。

5. 若在本毕业论文（设计）中有侵犯任何方面知识产权的行为，由本人承担相应的法律责任。

6. 本人完全了解**浙江大学**有权保留并向有关部门或机构送交本论文（设计）的复印件和磁盘，允许本论文（设计）被查阅和借阅。本人授权**浙江大学**可以将本论文（设计）的全部或部分内容编入有关数据库进行检索和传播，可以采用影印、缩印或扫描等复制手段保存、汇编本论文（设计）。

作者签名：　　导师签名：

签字日期：　　年　月　日　　签字日期　　年　月　日

致 谢

摘 要

Abstract

目录

第一部分

毕业论文

1 绪论

1.1 背景

1.1.1 节标题

2 Overleaf 使用注意事项

如果你在 Overleaf 上编译本模板，请注意如下事项[zjuthesis]：

- 删除根目录的 ".latexmkrc" 文件，否则编译失败且不报任何错误
- 字体有版权所以本模板不能附带字体，请务必手动上传字体文件，并在各个专业模板下手动指定字体。具体方法参照 GitHub 主页的说明。
- 当前（2019 年 9 月 2 日）的 Overleaf 使用 TexLive 2017 进行编译，但一些伪粗体复制乱码的问题需要 TexLive 2019 版本来解决。所以各位同学可以在 Overleaf 上编写论文，但务必使用本地的 TexLive 2019 来进行最终编译，以免产生查重相关问题。具体说明参照 GitHub 主页。

2.1 节标题

2.1.1 小节标题

我们可以用 includegraphics 来插入现有的 jpg 等格式的图片，如**??**。

图 2.1 浙江大学 LOGO

如**??**所示，这是一张自动调节列宽的表格。

如**??**，这是一个公式

表 2.1 自动调节列宽的表格

第一列	第二列
xxx	xxx
xxx	xxx
xxx	xxx

$$A = \overbrace{(a+b+c)+\underbrace{i(d+e+f)}_{\text{虚数}}}^{\text{复数}} \tag{2-1}$$

如**??**所示，这是一段代码。计算机学院的代码样式可能与其他专业不同，如有需要，可以从计算机学院专业模板中复制相关的代码样式设定。

代码 2.1: simple.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello, zjuthesis\n");
    return 0;
}
```

2.2 关于字体

英文字体通常提供了粗体和斜体的组合，中文字体通常没有粗体或斜体，本模板使用了 'AutoFakeBold' 来实现中文伪粗体，但不提供中文斜体，如**??**所示。

同一页上的章标题

表 2.2 一些字体示例

字体	常规	粗体	斜体	粗斜体
Times New Roman	Regular	**Bold**	*Italic*	***BoldItalic***
仿宋	常规	**粗体**	斜体	**粗斜体**
宋体	常规	**粗体**	斜体	**粗斜体**
黑体	**常规**	**粗体**	**斜体**	**粗斜体**
楷体	常规	**粗体**	斜体	**粗斜体**

附录

A　一个附录

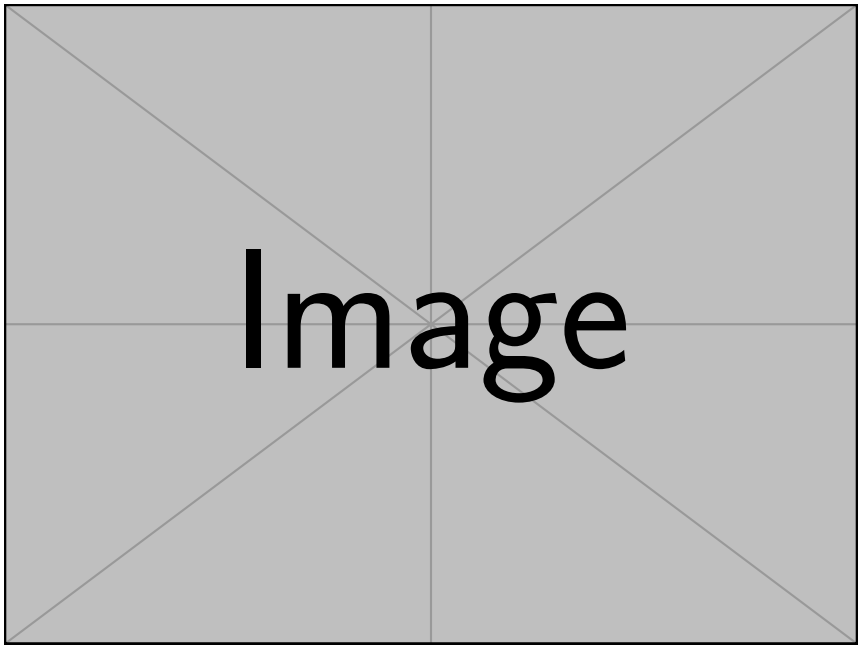

图 A.1 附录中的图片

$$E = mc^2 \tag{A-1}$$

B　另一个附录

$$x^n + y^n = z^n \tag{B-1}$$

作者简历

本科生毕业论文（设计）任务书

一、题目：

二、指导教师对毕业论文（设计）的进度安排及任务要求：

起讫日期 20　年　月　日　至　20　年　月　日

指导教师（签名）_____ 职称 _____

三、系或研究所审核意见：

负责人（签名）_____

年　月　日

本科生毕业论文（设计）考核

一、指导教师对毕业论文（设计）的评语：

指导教师（签名）_____

年　月　日

二、答辩小组对毕业论文（设计）的答辩评语及总评成绩：

成绩比例	文献综述（10%）	开题报告（15%）	外文翻译（5%）	毕业论文质量及答辩（70%）	总评成绩
分值					

负责人（签名）_____

年　月　日

第二部分

毕业论文开题报告

浙江大学

本科生毕业论文

文献综述和开题报告

学生姓名　　张乔

学生学号　　3200102817

指导教师　　纪守领

年级与专业　　2020级　计算机科学与技术

所在学院　　计算机科学与技术学院

一、题目：知识驱动的物联网嵌入式固件自动化根因分析方法研究

二、指导教师对文献综述和开题报告的具体要求：

文献综述：

1. 对嵌入式固件的根因分析的作用与重要性做出说明。

2. 调研当前学术界与工业界对根因分析和对嵌入式固件安全漏洞检测的研究现状。

3. 对目前该领域的研究现状进行分析，提出至少 3 项不足，并给出解决思路及未来可进行的研究方向。

4. 广泛开展调研，相关参考文献不少于 15 篇。

开题报告：

1. 对研究背景作详细说明，阐述研究的难点、意义与重要性。

2. 给出与其可行的技术路线，初步绘制系统架构图。

3. 合理规划研究进度。

4. 对预期研究成果给出量化可考核指标。

5. 广泛开展调查研究，相关参考文献不少于 15 篇，并注意参考文献的相关性、时效性与覆盖性。

指导教师（签名）

2024 年 3 月 10 日

目　录

目　录 3
一、文献综述 1
1．背景介绍 1
1.1 物联网嵌入式固件的模糊测试技术 1
1.2 自动化根因分析技术 2
2．国内外研究现状 3
2.1 研究方向及进展 3
2.1.1 基于事后分析的根因分析技术 3
2.1.2 基于频谱的根因分析技术 4
2.2 存在问题 5
3．研究展望 6
4．参考文献 6
二、开题报告 9
1．问题提出的背景 9
1.1 背景介绍 9
1.1.1 物联网嵌入式固件的模糊测试技术 9
1.1.2 自动化根因分析技术 10
1.2 本研究的意义和目的 11
2．论文的主要内容和技术路线 12
2.1 主要研究内容 12
2.2 技术路线 12
2.3 可行性分析 13
3．研究计划进度安排及预期目标 14
3.1 进度安排 14
3.2 预期目标 15
4．参考文献 15
三、外文翻译 18
摘要 18

1. 引言 19
2. 背景 22
2.1 American Fuzzy Lop (AFL) 22
2.2 MCU 固件分析 23
2.3 硬件支持的指令跟踪收集 23
2.3.1 追踪数据收集. 24
2.3.2 指令流重构 24
2.3.3 追踪数据过滤。 25
3. μAFL 设计 26
3.1 概述 26
3.2 低级设备控制和模糊测试调度 27
3.3 在线追踪收集器 27
3.3.1 测试用例传输 27
3.3.2 追踪收集和过滤 28
3.4 离线跟踪分析器 30
3.5 崩溃/挂起检测 33
4. 实现与评估 34
4.1 完全恢复指令流的开销 36
4.2 滤波器性能 36
4.3 开销细分 37
4.4 μAFL 整体性能 38
4.5 现实世界的固件模糊测试 40
5. 讨论 41
6. 相关工作 42
6.1 MCU 固件模糊测试 42
6.2 外设漏洞检测 43
7. 总结 43
四、外文原文 45
ABSTRACT 45
1. INTRODUCTION 46

2. BACKGROUND 49
2.1 American Fuzzy Lop (AFL) 49
2.2 Analysis of MCU Firmware 49
2.3 Hardware-Supported Instruction Trace Collection 50
3. μAFL DESIGN 52
3.1 Overview 52
3.2 Low-level Device Control and Fuzzing Scheduling 52
3.3 Online Trace Collector 53
3.4 Offline Trace Analyzer 55
3.5 Crash/Hang Detection 58
4. IMPLEMENTATION AND EVALUATION 59
4.1 Overhead of Fully Recovering Instruction Flow 60
4.2 Filter Performance 61
4.3 Overhead Breakdown 61
4.4 Overall Performance of μAFL 63
4.5 Real-world Firmware Fuzzing 64
5. DISCUSSIONS 65
6. RELATED WORK 66
6.1 MCU Firmware Fuzzing 66
6.2 Peripheral Vulnerability Detection 66
7. CONCLUSION 67
8. ACKNOWLEDGMENTS 67
9. REFERENCES 67
毕业论文（设计）文献综述和开题报告考核 69

一、文献综述

1. 背景介绍

随着硬件技术和网络通信技术的快速发展，越来越多的设备和系统被连接到互联网，实现了智能化的生活和工作环境。嵌入式固件是物联网设备的核心，它使得这些设备能够实现数据采集、通信、控制等功能，从而为用户提供更便捷、智能的服务体验。同时物联网技术的应用已经渗透到各个产业领域，包括国防、工业、家居、医疗、交通等行业。嵌入式固件在这些领域中起着关键作用，它使得设备和系统能够实现自动化、智能化的控制和管理，提高生产效率、节约资源、降低成本。

随着物联网设备的普及，信息安全和隐私保护越来越受到学术界与工业界的关注。嵌入式固件作为物联网设备的核心软件，需要具备良好的安全性和防护能力，以防止未经授权的访问、数据泄露、设备篡改等安全威胁，因此需要对其安全性进行检查。模糊测试技术可以通过将自动或半自动生成的测试输入到程序中，监测程序是否发生崩溃；而根因分析技术可以根据模糊测试的结果得到导致程序崩溃的测试输入，并寻找到程序中存在漏洞的位置。

本章将介绍物联网嵌入式固件的模糊测试技术与自动化根因分析技术。

1.1 物联网嵌入式固件的模糊测试技术

在安全测试领域，模糊测试是一种高效的测试技术。其核心思想是通过向系统输入大量自动生成的随机、异常或非预期的数据来检测输入验证不足、缓冲区溢出、代码注入等常见的安全漏洞和程序错误。在1988年，威斯康星大学的Barton Miller教授率先在他的课程实验提出模糊测试[5][6]。实验内容是开发一个基本的命令行模糊器以测试Unix程序。这个模糊器可以用随机数据来“轰炸”这些测试程序直至其崩溃。

目前的模糊测试工具如AFL[1]、AFL++[2]、Syzkaller[3]等已经在PC端发掘出应用程序与系统内核中的许多漏洞；μAFL[4]等已经在硬件领域发掘出许多0-day漏洞。嵌入式固件运行

在各种硬件平台上，并与多个复杂的外设通信，难以获取细粒度的反馈指导信息。因此，嵌入式固件模糊测试的主要挑战在于如何处理与外设的交互并收集内部运行时信息。一些研究者直接在真实的硬件设备上进行黑盒模糊测试[27]，缺乏详细的覆盖指导。μAFL[4]和SyzTrust[7]利用件调试器收集运行时信息，支持嵌入式跟踪宏单元(Embedded Trace Macrocell, ETM[8])功能。其他工作采用硬件在环方法[28]将硬件请求转发到真实的外围设备。然而，它们需要频繁地切换执行上下文并在模拟器和硬件之间同步状态，这导致了巨大的性能开销。黑盒模糊测试和硬件在环方法都依赖于实际硬件的可用性，并且由于代码覆盖信息很少，它们的后端模糊测试的能力受到限制。由于使用真正的硬件设备受到计算资源的限制，最近的研究采用了重托管技术来获得更好的可扩展性[26]。重托管的关键思想是对外设进行建模，并为固件提供有效的输入。这些输入主要是帮助固件在初始化阶段通过状态检查，导致它们与外设的实际输出不同，即在PC主机运行仿真器与模糊测试器，其中仿真器用于模拟物联网设备硬件环境，为固件提供运行环境；同时模糊测试器与仿真器进行交互，为固件提供测试样例，并从仿真器收集反馈信息。当前，QEMU[11]和 Unicorn[12]已经成为固件模糊测试工具所采用的事实标准仿真器。仿真过程基本上替代了对物理硬件的需求，使用户能够在受控的虚拟环境中分析甚至操纵固件代码。HALucinator[9]设计处理程序来取代硬件抽象层(Hardware Abstraction Layers, HAL)中的功能，实现仿真固件与主机环境之间的通信。Fuzzware[10]执行符号执行来自动生成外围模型，通过自动生成模型，帮助模糊测试过程将重点放在改变重要的输入上，从而大大提高其有效性。

1.2 自动化根因分析技术

在科学和工程学中，根因分析是一种解决问题的方法，用于识别故障或问题的根本原因。它广泛应用于IT运营、电信、工业过程控制、事故分析、医学等。根因分析通常是一种系统性的过程，需要收集和分析与问题相关的数据和信息，运用各种工具和技术来确定问题的根本原因。在计算机领域，特别是在软件安全领域，根因分析通常是指分析软件或系统崩溃的根本原因，它可能涉及对系统的代码、设计、配置、环境等方面进行深入分析。

通过根因分析，开发团队可以更好地了解程序中的潜在问题，并及时采取措施修复和加固系统，以提高软件的安全性和稳定性。

作为一种粗略的可靠性度量方法，模糊测试可以针对待分析程序产生不同输入得到程序崩溃，并提示程序哪些部件需要特殊的注意；而根因分析作为模糊模糊测试的后续阶段，可以利用，模糊测试得到的已知崩溃进行深入分析并找出导致这些问题的根本原因。

一般来说，当前自动化根因分析方法主要包括两类:基于频谱的根因分析方法和基于事后的根因分析方法。其中基于频谱的方法通常在初始崩溃测试用例的基础上生成两大组崩溃测试用例和非崩溃测试用例。它们记录每条指令上的寄存器和内存数据，以统计方式构建崩溃的必要条件作为谓词，测量每个谓词与初始崩溃的相关性，最后将谓词列为根本原因；而基于事后的故障定位方法从调试文件(例如核心转储和内存快照)开始，执行反向执行和向后污染分析，以跟踪无效数据的传播。它们分析指令的语义，比基于频谱的故障定位方法更有效。

2. 国内外研究现状

2.1 研究方向及进展

随着模糊测试等研究方法的发展，当前已经可以通过各种技术发现固件中的问题和漏洞，并得到能导致程序崩溃的各种输入，而根因分析技术则旨在利用相关信息进行深入分析并找到导致这些崩溃的根本原因。仅凭安全分析人员手动分析崩溃的根本原因是一项极其繁琐低效的工作，因此需要许多自动化的根本原因分析(Root Cause Analysis, RCA)系统来辅助手工分析。本章主要介绍目前自动化根因分析研究近况。

2.1.1 基于事后分析的根因分析技术

基于事后的根因分析技术是一种在崩溃发生后开始的根因分析技术。该方法的基本原理是通过收集和分析系统状态、日志、内存转储等信息，来确定问题的根本原因和触发条件。通过分析这些信息，可以确定程序的内存泄漏、空指针引用等异常行为，并采取措施

修复或预防类似问题的再次发生。在事后程序分析技术中，记录与重放[13,14,15]和核心转储分析[16,17,18]是两种常见方法。Manevich 等人[19]提出使用静态反向分析从崩溃点重构执行轨迹，从而发现软件缺陷，特别是类型状态错误[20]。类似地，Strom 和 Yellin[21]定义了一个部分路径敏感的反向数据流分析，用于检查类型状态属性，特别是未初始化的变量。Jun Xu[22]等人通过分析崩溃后的工件，引入了一种新的反向执行机制来构建程序崩溃之前的数据流，通过使用数据流辅助执行向后污染分析，并突出显示那些实际上导致崩溃的程序语句。

虽然当前这些研究被证明是有效的，但它们只关注特定的类型状态问题，或者在反向执行时难以解决内存别名等问题，仍然不足够高效。

2.1.2 基于频谱的根因分析技术

基于频谱的分析方法是一种利用软件系统的执行频谱（Execution Spectrum）来识别和定位问题的技术。该方法通常用于识别性能问题和资源利用不足等情况。基本原理是在软件运行时记录系统的执行轨迹或事件，然后通过分析这些事件的频率和模式来识别可能的瓶颈或问题。

例如，对于性能问题，可以记录系统中的函数调用、代码路径执行次数、资源利用情况等信息。通过分析这些数据，可以确定哪些函数或路径是性能瓶颈，并识别可能的优化方向。这种方法可以帮助开发人员了解系统的执行特征，并优化系统的性能和资源利用率。

大多数基于频谱的根因分析技术研究侧重于统计量的设计，包括排名指标和分布统计。但也有一些工作侧重对测试用例的研究：Hao 等人[23]根据测试用例的能力提出了三种减少测试用例数量的策略,基于使用测试输入的测试运行的执行跟踪，使开发人员可以只选择测试输入的一个有代表性的子集来进行结果检查和故障定位；Abreu 等人[24]通过使用由西门子集和空间程序组成的通用基准，研究了作为多个参数（例如系统执行期间收集的程序频谱的质量和数量）的函数的诊断准确性，结果表明用于分析程序谱的特定相似系数的优越性能在很大程度上独立于测试设计，并且证明了 SFL 可以有效地应用于工业环境中的嵌入式软件开发环境中；Dandan Xu 等人[25]设计并实现了一种反例强化学习技

术，该技术奖励涉及反例的操作，通过平衡随机抽样和对反例的利用，利用每个模糊测试回合的结果来指导下一轮模糊测试，从而将当今基于频谱的根因分析工具的可扩展性和准确性提高了一个数量级以上。

2.2 存在问题

目前对于模糊测试及后续的自动化根因分析研究已经取得了一定的进展，但是对于在资源受限硬件上运行的物联网嵌入式固件来说，现有工作仍然存在着一定的不足，具体来说，主要分以下三个方面。

缺少对物联网嵌入式固件的根因分析：为了定位导致程序崩溃的根本原因，分析人员或自动化根因分析工具在模糊测试后需要做大量工作，这对于物联网嵌入式固件场景来说，尤其困难，因为很难获取程序从开始到崩溃过程中的数据流和控制流。嵌入式固件运行在资源受限的设备上，因此缺乏像大型操作系统那样完善的日志机制比如 coredump 等。在故障发生时，缺乏足够的运行时信息，使得分析人员难以理解崩溃的根本原因。即使开发人员采用了一些简单的监控手段，也难以满足对故障进行有效分析的需求。因此当前工作缺少对物联网嵌入式固件有效的根因分析方法。

缺少对逆向执行时内存别名问题的解决技术：当需要逆向分析崩溃的执行轨迹时，分析人员面临内存别名问题。内存别名问题指的是在程序中存在多个指针，它们指向相同的内存位置或者相互之间存在重叠的内存区域。当存在内存别名时，对内存的读写操作可能会由多个指针引发，这增加了逆向执行时追踪程序状态的复杂性，同时内存别名也增加了程序本身的复杂性，使得理解程序行为变得更加困难。在数以万计的指令中手动追踪指针的传递和操作是一项繁琐耗时的任务。同时，采用现有算法进行内存别名识别，要么会涉及过多指令，不利于手动分析；要么会只涉及少量指令，无法定位远离崩溃点的根因。

缺少针对代码层次受限的执行环境的解决技术：为了满足低级硬件中受约束的计算资源的需求，嵌入式固件通常被剥离调试信息，调试信息通常包含了变量名、函数名、文件名、行号等关键信息，有助于理解代码的结构和逻辑。当这些信息被剥离时，只留下一堆混合着数据的连续指令，这使得分析人员将难以理解代码的语义，从而难以执行上下文，从而导致根因分析变得更加困难。同时，调试信息有助于在代码中追踪程序执行的路径，

包括函数调用关系、条件分支情况等。缺乏这些信息会使得分析人员难以跟踪程序的执行流程，增加了根因分析的复杂性。在这种情况下，分析人员通常依赖后期故障定位方法或基于频谱的故障定位方法，但这些方法并不能提供足够的指导，使得分析工作更加困难。

3. 研究展望

根据当前模糊测试技术与自动化根因分析技术的学习研究，提出了三项可能的研究展望。

基于人工智能模型方法的模糊测试方法。在模糊测试产生不同输入的过程大多是随机的，可以利用人工智能模型利用每个模糊测试回合的结果，包括其对每个谓词对崩溃的估计贡献的影响，以及基于其贡献的这些谓词的估计排名，来指导下一轮模糊测试，影响种子选择和突变算子与位置的选择，更有效地产生崩溃发现漏洞，从而提高其在查找漏洞根本原因方面的有效性。

基于历史信息收集的根因分析方法。对于物联网固件场景来说，很难获取崩溃过程中数据流和控制流，并且由于内存别名等问题，当前很难在合理的时间预算内分析长执行轨迹。可以在执行导致崩溃的输入时，采取特定的监视和有效信息收集方法，手机可以极大提高逆向执行效率的执行中信息，用于辅助后续的根因分析，从而较大提高根因分析的效率和能力。

基于大语言模型辅助的根因分析方法。大语言模型可以分担较多的机械重复性的手动工作，可以用于较大文件的审查，快速识别潜在的问题和错误。它们可以分析日志等较大的文件或其他信息，发现其中的异常，为分析人员提供指导和建议。同时基于大语言模型的自然语言生成能力，可以自动生成解释文档和建议，帮助分析人员理解和解释复杂的系统行为，提供根因分析的指导和建议。

4. 参考文献

[1] Michal Zalewski. 2010. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[2] Fioraldi, Andrea, et al. "{AFL++}: Combining incremental steps of fuzzing research." 14th USENIX Workshop on Offensive Technologies (WOOT 20). 2020.

[3] VYKOV D， KONOVALOV A. Sykaller: an unsupervised, coverage-guided kernel fuzzer [Z].2019

[4] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan.2022. μAFL: non-intrusive feedback-driven fuzzing for microcontroller firmware.In Proceedings of the 44th International Conference on Software Engineering. 1–12.

[5] ARM. Barton Miller (2008). "Preface". In Ari Takanen, Jared DeMott and Charlie Miller, Fuzzing for Software Security Testing and Quality Assurance, ISBN 978-1-59693-214-2

[6] Fuzz Testing of Application Reliability. University of Wisconsin-Madison. [2009-05-14].

[7] Qinying Wang, Boyu Chang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Gaoning Pan, Chenyang Lyu, Mathias Payer, Wenhai Wang, and Raheem Beyah. 2024. SyzTrust: State-aware Fuzzing on Trusted OS Designed for IoT Devices. In 2024 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, 70–70.

[8] ARM. 2011. Embedded Trace Macrocell, ETMv1.0 to ETMv3.5, Architecture Specification. https://documentation-service.arm.com/static/5f90158b4966cd7c95fd5b5e.

[9] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware re-hosting through abstraction layer emulation. In 29th USENIX Security Symposium (USENIX Security 20). 1201–1218.

[10] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench,Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. 2022. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In 31st USENIX Security Symposium (USENIX Security 22). 1239–1256.

[11] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In USENIX annual technical conference, FREENIX Track, Vol. 41. Califor-nia, USA, 46.

[12] Unicorn Engine. [n. d.]. The Ultimate CPU emulator．https://www.unicornengine.org/

[13] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In Proceedings of the 22Nd European Conference on ObjectOriented Programming, 2008.

[14] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. In Proceedings of the 2013 International Conference on Software Engineering, 2013

[15] Y. Cao, H. Zhang, and S. Ding. Symcrash: Selective recording for reproducing crashes. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, 2014.

[16] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In Proceedings of the 38th International Conference on Software Engineering, 2016.

[17] P. Ohmann. Making your crashes work for you (doctoral symposium). In Proceedings of the 2015 International Symposium on Software Testing and Analysis, 2015.

[18] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu. Credal: Towards locating a memory corruption vulnerability with your core dump. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016.

[19] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. Pse: Explaining program failures via postmortem static analysis. In Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, 2004.

[20] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. IEEE Transaction Software Engineering, 1986.

[21] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. IEEE Transaction Software Engineering, 1993.

[22] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem program analysis with Hardware-Enhanced Post-Crash artifacts. In 26th USENIX Security Symposium (USENIX Security 17). 17–32.

[23] Dan Hao, Tao Xie, Lu Zhang, Xiaoyin Wang, Jiasu Sun, and Hong Mei. Test input reduction for result inspection to facilitate fault localization. Automated software engineering, 17:5–31, 2010.

[24] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software, 82(11):1780–1792, 2009.

[25] Dandan Xu, Di Tang , Yi Chen, XiaoFeng Wang, Kai Chen, Haixu Tang, and Longxing Li. Racing on the Negative Force: Efficient Vulnerability Root-Cause Analysis through Reinforcement Learning on Counterexamples. In 33rd USENIX Security Symposium.

[26] WRIGHT C, MOEGLEIN W A, BAGCHI S, et al. Challenges in firmware re-hosting, emulation, and analysis [J]. ACM Computing Surveys (CSUR), 2021, 54(1): 1-36.

[27] Feng, Xiaotao, et al. "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference." Proceedings of the 2021 ACM SIGSAC conference on computer and communications security. 2021.

[28] Ahmadi-Pour, Sallar, Pascal Pieper, and Rolf Drechsler. "Virtual-Peripheral-in-the-Loop: A Hardware-in-the-Loop Strategy to Bridge the VP/RTL Design-Gap." arXiv preprint arXiv:2311.00442 (2023).

二、开题报告

1. 问题提出的背景

1.1 背景介绍

1.1.1 物联网嵌入式固件的模糊测试技术

在安全测试领域，模糊测试是一种高效的测试技术。其核心思想是通过向系统输入大量自动生成的随机、异常或非预期的数据来检测输入验证不足、缓冲区溢出、代码注入等常见的安全漏洞和程序错误。目前的模糊测试工具如AFL[1]、AFL++[2]、Syzkaller[3]等已经在PC端发掘出应用程序与系统内核中的许多漏洞；μAFL[4]等已经在硬件领域发掘出许多0-day漏洞。对于物联网设备而言，其系统内存资源极为有限，使得传统的模糊测试方法难以直接应用。目前对于物联网设备进行模糊测试的方法主要采取重新托管的方法，即在PC主机运行仿真器与模糊测试器，其中仿真器用于模拟物联网设备硬件环境，为固件提供运行环境；同时模糊测试器与仿真器进行交互，为固件提供测试样例，并从仿真器收集反馈信息。

嵌入式固件运行在各种硬件平台上，并与多个复杂的外设通信，难以获取细粒度的反馈指导信息。因此，嵌入式固件模糊测试的主要挑战在于如何处理与外设的交互并收集内部运行时信息。一些研究者直接在真实的硬件设备上进行黑盒模糊测试[27]，缺乏详细的覆盖指导。μAFL[4]和SyzTrust[7]利用件调试器收集运行时信息，支持嵌入式跟踪宏单元(Embedded Trace Macrocell, ETM[8])功能。其他工作采用硬件在环仿真方法[28]将硬件请求转发到真实的外围设备。然而，它们需要频繁地切换执行上下文并在模拟器和硬件之间同步状态，这导致了巨大的性能开销。黑盒模糊测试和硬件在环方法都依赖于实际硬件的可用性，并且由于代码覆盖信息很少，它们的后端模糊测试的能力受到限制。由于使用真正的硬件设备受到计算资源的限制，最近的研究采用了重托管技术来获得更好的可扩展性[26]。重托管的关键思想是对外设进行建模，并为固件提供有效的输入。这些输入主要是帮助固

件在初始化阶段通过状态检查，导致它们与外设的实际输出不同。HALucinator[9]设计处理程序来取代硬件抽象层(Hardware Abstraction Layers, HAL)中的功能，实现仿真固件与主机环境之间的通信。Fuzzware[10]执行符号执行来自动生成外围模型，通过自动生成模型，帮助模糊测试过程将重点放在改变重要的输入上，从而大大提高其有效性。

固件重托管是一种在模拟环境中执行固件二进制文件的技术，在仿真器中为物联网设备固件提供模拟硬件环境，包括内存和寄存器资源、虚拟外设支持以及 CPU 架构的仿真。当前，QEMU[11]和 Unicorn[12]已经成为固件模糊测试工具所采用的事实标准仿真器。仿真过程基本上替代了对物理硬件的需求，使用户能够在受控的虚拟环境中分析甚至操纵固件代码。

1.1.2 自动化根因分析技术

在科学和工程学中，根因分析是一种解决问题的方法，用于识别故障或问题的根本原因。它广泛应用于IT运营、电信、工业过程控制、事故分析、医学等。根因分析通常是一种系统性的过程，需要收集和分析与问题相关的数据和信息，运用各种工具和技术来确定问题的根本原因。在计算机领域，特别是在软件安全领域，根因分析通常是指分析软件或系统崩溃的根本原因，它可能涉及对系统的代码、设计、配置、环境等方面进行深入分析。通过根因分析，开发团队可以更好地了解程序中的潜在问题，并及时采取措施修复和加固系统，以提高软件的安全性和稳定性。

作为一种粗略的可靠性度量方法，模糊测试可以针对待分析程序产生不同输入得到程序崩溃，并提示程序哪些部件需要特殊的注意；而根因分析作为模糊模糊测试的后续阶段，可以利用，模糊测试得到的已知崩溃进行深入分析并找出导致这些问题的根本原因。

一般来说，当前自动化根因分析方法主要包括两类:基于频谱的根因分析方法和基于事后的根因分析方法。其中基于频谱的方法通常在初始崩溃测试用例的基础上生成两大组崩溃测试用例和非崩溃测试用例。它们记录每条指令上的寄存器和内存数据，以统计方式构建崩溃的必要条件作为谓词，测量每个谓词与初始崩溃的相关性，最后将谓词列为根本原因；而基于事后的故障定位方法从调试文件(例如核心转储和内存快照)开始，执行反向

执行和向后污染分析，以跟踪无效数据的传播。它们分析指令的语义，比基于频谱的故障定位方法更有效。

1.2 本研究的意义和目的

物联网已经逐步进入社会的生活生产中，如基于阿里云物联网平台搭建的新一代智能光伏发电系统，近百万台光伏逆变器上云，整体提升光伏发电效率，实时远程控制及持续分析数据，大幅降低人力维护成本。嵌入式固件作为物联网设备的核心软件，需要具备良好的安全性和防护能力，以防止未经授权的访问、数据泄露、设备篡改等安全威胁，因此需要对其安全性进行检查。

当前，模糊测试技术可以通过将自动或半自动生成的测试输入到程序中，监测程序是否发生崩溃；而根因分析技术可以根据模糊测试的结果得到导致程序崩溃的测试输入，并寻找到程序中存在漏洞的位置。然而，目前尚未有针对运行资源受限的物联网嵌入式固件的自动化根因分析方法，无法针对嵌入式固件崩溃进行深入分析。同时为了满足低级硬件中受约束的计算资源的需求，嵌入式固件通常被剥离调试信息，调试信息通常包含了变量名、函数名、文件名、行号等关键信息，有助于理解代码的结构和逻辑。当这些信息被剥离时，只留下一堆混合着数据的连续指令，这使得分析人员将失去对代码执行上下文的理解，更难理解代码的语义，从而导致手动根因分析变得更加困难。同时由于逆向执行时存在内存别名等问题，同时难以得到程序运行时的控制流与数据流，传统的自动化根因分析工具难以生效。

综上所述，针对物联网嵌入式固件设备，尚未存在有效的自动化根因分析方法衔接模糊测试，无法确定固件发生崩溃崩溃的根本原因并加以修正。因此，设计针对嵌入式固件的自动化根因分析工具是当前急需解决的问题。

本研究围绕物联网嵌入式固件的根因分析这一主题，总结分析其所面临的安全挑战，凝练出两个关键科学问题：如何解决物联网嵌入式固件仅有有限的计算资源和存储空间难以运行现有的操作系统程序自动化根因分析方法、如何解决难以获取数据流控制流等运行时信息导致难以高效逆向执行的问题。研究希望通过解决这两个难题，提出针对物联网嵌

入式固件的根因分析方法，同时融合已有的模糊测试方法，提出对嵌入式固件的安全检查体系。

2. 论文的主要内容和技术路线

2.1 主要研究内容

面向物联网嵌入式固件模糊测试结果的信息收集。物联网嵌入式固件的计算资源和存储空间受限的特点，针对其难以实现高覆盖率和高效率的问题，本研究拟提出基于仿真的自动化根因分析信息收集方法。在仿真环境中，可以模拟不同类型和规模的硬件设备，而不受真实硬件的限制，这使得可以更灵活地测试和验证固件的功能和性能。仿真过程基本上替代了对物理硬件的需求。因此，在资源充足且受控的虚拟环境中仿真模拟运行物联网嵌入式固件，可以分配充足的运算资源和存储资源，记录后续分析所需的信息，并加以存储。

基于仿真运行和必要信息收集的物联网嵌入式固件高效逆向执行。为解决物联网嵌入式固件的计算资源和存储空间受限下难以高效进行逆向执行的难题，本研究拟提出基于仿真信息的逆向执行技术。通过仿真运行中收集的信息构造使用-定义链，并恢复程序执行中的控制流和数据流。在此基础上，在后向污点分析中，通过将通用寄存器视为漏洞点，在反向传播的过程中，通过查找前面构造的使用-定义链，识别污变量。由此实现高效的逆向执行工作。

2.2 技术路线

考虑到目前物联网嵌入式固件的计算资源和存储空间受限的特点，针对其难以实现高覆盖率和高效率的问题，本研究提出基于仿真的自动化根因分析信息收集方法，在仿真运行过程中利用充足资源记录后续自动化分析所需要的相关信息，并在后续逆向执行时利用前期收集的信息恢复数据流和控制流，辅助进行后向污点分析。

为解决传统物联网设备难以获取内部信息、不能有效指导根因分析进行的问题，本研究拟通过仿真执行方式收集系统内核执行信息，实现内核代码运行的高效监测。在此基础上通过重建控制流和数据流，大大提高逆向执行的效率。

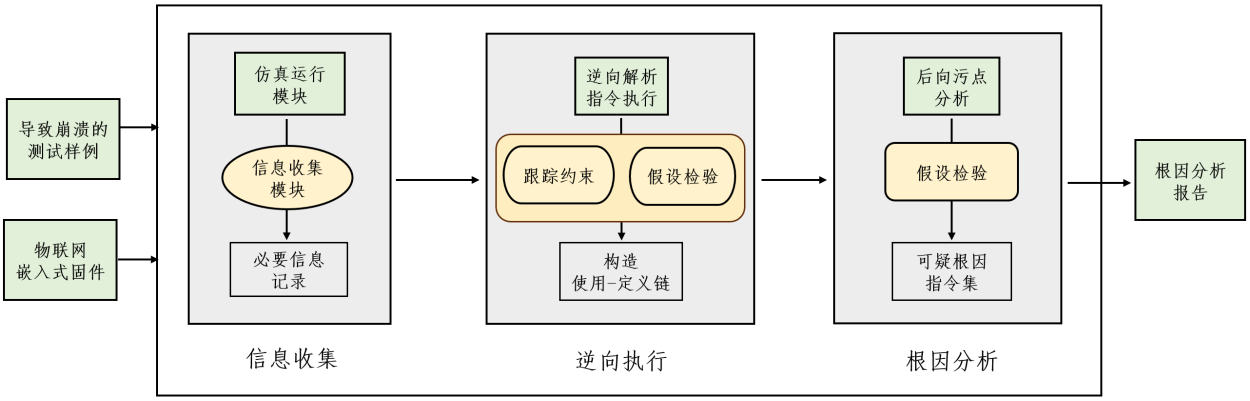

图 1：本研究整体架构图。本工作将模糊测试中发现的导致崩溃的输入和物联网嵌入式固件本身当作输入；通过仿真重现崩溃并记录必要信息，反向执行时，利用信息构造使用定义链条，最后执行后向污点分析，得到根因指令集。

整体架构如图 1 所示。本研究计划使用 Unicorn 进行作为硬件仿真支持，使用由模糊测试工具生成的崩溃测试用例和相应的固件二进制文件作为输入，并再现崩溃以收集足够的运行时数据，包括指令执行地址、崩溃现场寄存器信息等。之后利用这些运行时信息来执行反向执行并构建一个使用定义链，并沿着使用定义链执行反向污染分析，以识别与崩溃有关的指令。最终，形成一个根因分析报告。

2.3 可行性分析

研究主题与研究方案可行。在大型操作系统下，可以利用核心转储信息或 Intel-PT 等硬件辅助处理器跟踪技术来跟踪程序执行过程，从而获得充分的执行时信息，这些信息会极大提高后续分析的效率。而已有的自动化根因分析工具难以在物联网嵌入式固件上有效工作。在模糊测试发现漏洞后，安全分析人员需要进行大量手动工作以找到导致崩溃的根本原因，同时由于缺少调试信息，内存别名等问题，当前对物联网嵌入式固件则缺少自动

化的根因分析技术。尽管如此，可以设计一些方法，在程序执行时记录少量必要信息，在后续分析时利用有限的运行时信息，采取假设检验等技术来实现根因分析自动化和高效化。

技术上具备可行性。目前已经存在较为成熟的模糊测试技术和自动化根因分析技术，同时已有手动分析方法，这些技术可以作为对物联网嵌入式固件进行自动化根因分析的基础。因此，在技术上实现物联网嵌入式固件的自动化根因分析并建立嵌入式固件安全检查体系是可行的。

具备相关研究条件。进行自动化根因分析需要充足的计算资源，包括开发工具、建立测试环境、收集数据集等。现阶段已收集完相关测试用例，并已获得到相关固件二进制文件。经初步仿真平台测试，系统运行基本符合预期，且实验室经费、服务器等科研资源充足，能充分支持后续各项研究。

3．研究计划进度安排及预期目标

3.1 进度安排

2024.3.1-2024.3.15：部署仿真环境初步测试。在此基础上阅读相关文献工作，进一步深入了解物联网嵌入式固件模糊测试机制。

2024.3.16-2024.3.31：完成崩溃后程序信息收集模块，并对测试样例进行测试，作为后续实验的对照。完成反向执行模块设计，进行初步实验验证。

2024.4.1-2024.4.20：根据收集到的信息进行反向执行，并选择部分测试样例进行手动分析检验效果。同时融合已有的面向物联网嵌入式固件模糊测试方法，形成一个安全性检查的体系。

2024.4.21-2024.5.20：分析实验结果，优化实验方案。完成论文撰写、专利申请等工作。

3.2 预期目标

本研究围绕物联网嵌入式固件领域，面向其安全性分析，从崩溃后的程序收集信息并进行逆向执行以以完成根因分析工作，并有效融合模糊测试方法，提出一个有效面向物联网嵌入式固件的安全检查体系。具体而言，包括：

a）研制针对物联网嵌入式固件的自动化根因分析方法，使分析效率有所保障，针对其他嵌入式固件的手动根因分析方法能力和效率均有显著提升。

b）可以较好衔接已有的面向嵌入式固件模糊测试工作，形成物联网嵌入式固件的安全检查体系，并对智能驾驶、智能家居等至少 3 个使用环境下的的嵌入式固件进行检查，提供检测报告。

c）申请发明专利 1 项。

4. 参考文献

[1] Michal Zalewski. 2010. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[2] Fioraldi, Andrea, et al. "{AFL++}: Combining incremental steps of fuzzing research." 14th USENIX Workshop on Offensive Technologies (WOOT 20). 2020.

[3] VYKOV D， KONOVALOV A. Sykaller: an unsupervised, coverage-guided kernel fuzzer [Z].2019

[4] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan.2022. μAFL: non-intrusive feedback-driven fuzzing for microcontroller firmware.In Proceedings of the 44th International Conference on Software Engineering. 1–12.

[5] ARM. Barton Miller (2008). "Preface". In Ari Takanen, Jared DeMott and Charlie Miller, Fuzzing for Software Security Testing and Quality Assurance, ISBN 978-1-59693-214-2

[6] Fuzz Testing of Application Reliability. University of Wisconsin-Madison. [2009-05-14].

[7] Qinying Wang, Boyu Chang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Gaoning Pan, Chenyang Lyu, Mathias Payer, Wenhai Wang, and Raheem Beyah. 2024. SyzTrust: State-aware Fuzzing on Trusted OS Designed for IoT Devices. In 2024 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, 70–70.

[8] ARM. 2011. Embedded Trace Macrocell, ETMv1.0 to ETMv3.5, Architecture Specification. https://documentation-service.arm.com/static/5f90158b4966cd7c95fd5b5e.

[9] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware re-hosting through abstraction layer emulation. In 29th USENIX Security Symposium (USENIX Security 20). 1201–1218.

[10] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench,Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. 2022. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In 31st USENIX Security Symposium (USENIX Security 22). 1239–1256.

[11] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In USENIX annual technical conference, FREENIX Track, Vol. 41. Califor-nia, USA, 46.

[12] Unicorn Engine. [n. d.]. The Ultimate CPU emulator. https://www.unicornengine.org/

[13] W Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. 2012. Software fault localization using dstar (d*). In 2012 IEEE Sixth International Conference on Software Security and Reliability. IEEE, 21–30

[14] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. IEEE Transaction Software Engineering, 1986.

[15] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. IEEE Transaction Software Engineering, 1993.

[16] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem program analysis with Hardware-Enhanced Post-Crash artifacts. In 26th USENIX Security Symposium (USENIX Security 17). 17–32.

[17] Dan Hao, Tao Xie, Lu Zhang, Xiaoyin Wang, Jiasu Sun, and Hong Mei. Test input reduction for result inspection to facilitate fault localization. Automated software engineering, 17:5–31, 2010.

[18] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software, 82(11):1780–1792, 2009.

[19] Dandan Xu, Di Tang , Yi Chen, XiaoFeng Wang, Kai Chen, Haixu Tang, and Longxing Li. Racing on the Negative Force: Efficient Vulnerability Root-Cause Analysis through Reinforcement Learning on Counterexamples. In 33rd USENIX Security Symposium.

[20] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. IEEE Transaction Software Engineering, 1986.

[21] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. IEEE Transaction Software Engineering, 1993.

[22] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem program analysis with Hardware-Enhanced Post-Crash artifacts. In 26th USENIX Security Symposium (USENIX Security 17). 17–32.

[23] Dan Hao, Tao Xie, Lu Zhang, Xiaoyin Wang, Jiasu Sun, and Hong Mei. Test input reduction for result inspection to facilitate fault localization. Automated software engineering, 17:5–31, 2010.

[24] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software, 82(11):1780–1792, 2009.

[25] Dandan Xu, Di Tang , Yi Chen, XiaoFeng Wang, Kai Chen, Haixu Tang, and Longxing Li. Racing on the Negative Force: Efficient Vulnerability Root-Cause Analysis through Reinforcement Learning on Counterexamples. In 33rd USENIX Security Symposium.

[26] WRIGHT C, MOEGLEIN W A, BAGCHI S, et al. Challenges in firmware re-hosting, emulation, and analysis [J]. ACM Computing Surveys (CSUR), 2021, 54(1): 1-36.

[27] Feng, Xiaotao, et al. "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference." Proceedings of the 2021 ACM SIGSAC conference on computer and communications security. 2021.

[28] Ahmadi-Pour, Sallar, Pascal Pieper, and Rolf Drechsler. "Virtual-Peripheral-in-the-Loop: A Hardware-in-the-Loop Strategy to Bridge the VP/RTL Design-Gap." arXiv preprint arXiv:2311.00442 (2023).

[29] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. Statistical debugging: A hypothesis testing-based approach. IEEE Transactions on software engineering, 32(10):831–848, 2006.

[30] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, WeiHan Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In 28th USENIX Security Symposium (USENIX Security 19), pages 1949–1966, 2019.

[31] Wes Masri. Fault localization based on information flow coverage. Software Testing, Verification and Reliability, 20(2):121–147, 2010.

三、外文翻译

微控制器固件的非侵入式反馈驱动模糊测试

Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware

摘要

模糊测试是挖掘软件漏洞最有效的方法之一。然而，将其应用于微控制器固件会带来许多挑战。例如，基于重新托管的解决方案无法准确建模外围设备行为，因此无法用于模糊测试相应的驱动程序代码。在本文工作中，我们提出了 μAFL，一种用于模糊测试微控制器固件的硬件在环方法。它利用现有嵌入式系统开发的调试工具来构建与 AFL 兼容的模糊测试框架。我们使用调试转接器来桥接 PC 上的模糊测试环境和微控制器设备上的目标固件。为了收集代码覆盖率信息而无需昂贵的代码插装，μAFL 依赖于 ARM ETM 硬件调试功能，该功能可透明地收集指令跟踪并将结果流式传输到 PC。然而，原始的 ETM 数据是难以理解的，并且需要巨大的计算资源来恢复实际的指令流。因此，我们提出了代码覆盖的另一种表示形式，虽然保留了原始 AFL 算法的相同路径敏感性，但可以直接处理原始 ETM 数据，而无需将其与反汇编的指令进行匹配。为了进一步减轻工作量，我们使用了 DWT 硬件特性来选择性地收集感兴趣的运行时信息。我们在两个真实评估板上对 μAFL 进行了评估。通过我们的原型，我们在 STMicroelectronics 的 SDK 中发现了十个 0-day 漏洞，在 NXP 的 SDK 中发现了三个 0-day 漏洞，并为它们分配了八个 CVE 编号。考虑到供应商 SDK 在实际产品中的广泛采用，我们的结果是令人担忧的。

关键词： 固件安全、模糊测试、微控制器、物联网（IoT）

1. 引言

物联网（IoT）已经成为我们数字生活中不可或缺的一部分。例如，许多人安装智能恒温器以远程控制家中的温度和湿度。智能报警系统用于监视家庭和工作场所，并在检测到入侵者时发出警报。健身追踪器和智能健康手环也被广泛用于持续监测个人健康数据，如心率和血氧水平。许多物联网设备的关键组件是微控制器单元（MCU），它是一种微小、定制和成本效益高的片上系统（SoC）[26]。

微控制器生态系统的快速演变一方面使我们的生活比以往任何时候都更加轻松和便利，另一方面也在野外引入了大量易受攻击的 MCU 产品。例如，最近针对 ESP8266 和 ESP32 通信和 WiFi 协处理器报告了几个备受关注的漏洞，这些漏洞（EAP client crash、zero PMK installation 和 beacon frame crash [35]）允许对手劫持或崩溃 ESP32/ESP8266 产品的会话。同时，FreeR-TOS TCP/IP 协议栈的漏洞可以用于发起远程代码执行和窃取私人信息[48,49]。更近期的 BadAlloc[9]使大量物联网设备暴露在攻击者面前。MCU 产品的妥协可能导致严重后果，如隐私泄露、财务损失，甚至人员伤亡。为了防止或减轻此类攻击，开发阶段进行广泛的安全测试是必不可少的。

然而，现有的固件安全测试方法在应用于嵌入式固件测试时展现出了各自的局限性（参见表 1，概述了这些问题）。例如，基于仿真的重新托管技术在近年来在固件分析方面得到了广泛研究[11,13,17,19,22,27,47]，但准确建模多样化外围设备的行为仍然是主要的研究挑战。几种方法，如 P2IM[17]、DICE[27]、Lae-laps[11]、PRETENDER[19]、μEmu[47]和 Jetset[22]，提出使用符号执行、访问模式匹配和机器学习技术学习近似的外围设备模型。然而，这些学习模型通常是不准确的。由于模型不准确，这种重新托管方法无法引导带有 USB 等复杂外围设备的固件。即使固件能够“成功”引导，但当执行跟真实设备上的执行跟踪不完全相同时，对于许多安全分析任务来说，这是不足够的。

表 1：与最先进的解决方案的比较

Basic Method	Solutions	Hardware Independent Code	Driver Code	Support Fuzzing	Require Source Code	Require Hardware
Rehosting	HALucinator [13], P^{2}IM [17], μEmu [47], etc.	✓	✗	✓	N	N
Porting	Para-rehosting [43]	✓	✗	✓	Y	N
Forwarding Hardware Interactions	Avatar [45], Avatar^2 [30], SURROGATES [24], Inception [14], etc.	✓	✗*	✗*	N	Y
Fully On-device Execution	***μAFL* (proposed)**	✓	✓	✓	N	Y

*:理论上，这些解决方案支持模糊驱动程序代码。然而，状态同步的巨大开销使得模糊驱动程序代码变得不切实际。现有解决方案利用真实设备将 QEMU 中的固件启动到可以分析与硬件无关的代码的状态。

基于仿真的重新托管技术通常用于在目标固件通过引导过程后测试硬件独立代码。

另一个方向是利用微控制器（MCU）固件中可用的硬件抽象层（HAL）来避免建模外围设备。例如，HALu-cinator[13]自动检测 HAL 库并将其替换为主机实现。Para-rehosting[43]提供了常见的 HAL 后端实现，以帮助将 MCU 固件移植到本地主机。然而，这两种方法都无法测试从未运行的外围设备驱动程序。

最后，一些硬件在环（HITL）解决方案在 QEMU 仿真器中运行固件，同时将外围 I/O 操作转发到真实设备[14,24,30,45]。尽管保留了高度的准确性,但这些基于 HITL 的方法需要在 QEMU 和硬件之间进行频繁且昂贵的切换（和同步），导致显著的性能开销。例如，当需要频繁进行硬件交互时，仿真速度约为每秒十几条指令[45]。因此，这些方法通常用于在固件在借助转发机制完全引导后，分析硬件独立代码。据我们所知，目前没有任何现有的 HITL 工作支持固件模糊测试。

在本工作中，我们提出了一种新的模糊测试解决方案，称为微型 AFL（μAFL），专门为基于 ARM 的 MCU 设备设计，以帮助开发人员在固件中定位潜在的软件漏洞。虽然我们的方法可以覆盖整个软件堆栈，但在本文中，我们将重点放在其在测试低层代码（如外围设备驱动程序）中的使用，因为我们认为现有工作没有很好地支持这一点。通过直接在目标设备上运行目标固件，我们的方法支持高度保真度的全栈测试。μAFL 需要开发人员可以访问原型开发板和基于 JTAG 或 SWD 的调试转接器，这些是几乎每个嵌入式系统开发环境中都可用和使用的基本硬件工具。

μAFL 被设计为模块化和可扩展的，以便可以直接集成现有成熟的模糊测试器（例如，我们在原型实现中采用了 AFL[46]，但任何其他模糊测试器都可以用作直接替换）。为了实现这一目标，(i) μAFL 将执行引擎与模糊测试器的其余部分（本文统称为模糊测试管理器）

解耦。具体来说，我们在真实硬件上运行目标固件，并在协调模糊测试过程的 PC 上运行模糊测试管理器，以调试转接器的帮助。执行信息也被传送到模糊测试管理器，在那里进行分析。通过这种方式，我们保持了 PC 上的模糊测试管理器对执行引擎的不可知性。(ii)为了使开发板与 PC 之间能够通信，我们重新利用了调试转接器，它对开发板具有最高级别的控制。因此，μAFL 可以高效有效地将测试案例传输到开发板上，提取运行时执行状态，并启动/挂起/停止目标。(iii)最后，为了收集每个测试案例的代码覆盖信息，这对于灰盒反馈驱动的模糊测试器至关重要，μAFL 利用了称为嵌入式跟踪宏单元(ETM)的硬件特性[4]，它可以透明地生成指令跟踪。跟踪结果通过五个额外的引脚（四个用于数据，一个用于时钟）直接传输到 PC。尽管 ETM 会增加额外成本，但我们在这项工作中采用它有两个原因。首先，它实现了透明的跟踪收集。换句话说，不需要代码插装。这个特性使我们的工作免受重写二进制文件的影响，因为大多数第三方库都是作为剥离二进制文件分发的，且没有可靠的二进制重写工具可用于促进基于插装的跟踪收集。其次，我们认为只有在开发阶段才需要具有 ETM 引脚的原型开发板。固件经过充分测试后，发布的产品就不需要配备这些功能。考虑到后期可能发生的昂贵召回，这种在开发阶段的小投资长期来看会有很好的回报。

μAFL 有两个关键组件：在线追踪收集器和离线追踪分析器。它们分别在设备上收集 ETM 数据，然后在 PC 上解析结果。当在 PC 上有一个由 AFL 生成的测试用例时，μAFL 通过调试连接器将其发送到板子上的保留内存中。在第一次消耗测试用例的时候，在线跟踪收集器会激活 ETM 来收集指令追踪数据，并将数据流传输到个人电脑。在收集 ETM 数据流的同时，在线跟踪收集器还通过数据监视和跟踪（DWT）单元[2]应用可配置的过滤器来抑制不必要的 ETM 数据包生成。这不仅减少了传输的追踪数据量，还避免了个人电脑上分析无用数据包。

离线跟踪分析器在个人电脑上运行，并处理原始的 ETM 数据。结果提供给 AFL 以维护代码覆盖率的位图。解码原始的 ETM 数据以获取分支信息是昂贵的，因为它需要反汇编固件并将指令与原始跟踪对齐[18]（请参见 RQ1 在第 4 节中）。我们通过在运行时使用一种特殊的基本块来解决这个问题。这使我们能够直接使用原始的 ETM 数据而无需反汇编，但仍

保留了需要计算代码覆盖率的路径敏感性。离线跟踪分析器还使用基于软件的方法来过滤在线跟踪收集器无法过滤的无趣的 ETM 数据包。

我们使用 SEGGER MCU 调试解决方案[38]实现了μAFL 的原型。然后，我们使用该原型测试了来自两家主要 MCU 芯片供应商的 SDK，即 NXP Semicon-ductors[34]和 STMicroelectronics[42]。特别地，我们将 USB 驱动程序模糊测试作为我们评估的案例研究。截至撰写本文时，我们发现了先前未知的 13 个驻留在 USB 驱动程序中的错误。所有这些错误都已经被供应商确认，并且随着最新的 SDK 发布或计划发布了补丁。

总的来说，我们的贡献有三个方面：

- 我们提出了μAFL，这是第一个适用于 MCU 固件驱动代码的模糊测试工具。μAFL 将执行引擎与模糊测试管理器分离，以便现有的模糊测试工具可以轻松集成。
- 我们提出使用 ARM ETM 进行非侵入式反馈收集。为了提高性能，μAFL 采用线性代码序列和跳转（LCSAJ）分析，直接处理原始 ETM 数据，而无需昂贵的反汇编。
- 我们已经实现并评估了我们的原型，针对主要 MCU 芯片厂商的两个 SDK。我们以 USB 驱动程序作为案例研究，展示了我们的原型如何对真实世界的驱动程序代码进行模糊测试。该工具帮助我们发现了 13 个以前未知的漏洞，并分配了 8 个 CVE 编号。

评估中使用的源代码和固件样本可在 https://github.com/MCUSec/microAFL 上获取，供今后在这个主题上进行研究使用。

2. 背景

2.1 American Fuzzy Lop (AFL)

模糊测试是一种自动化测试技术，用于发现软件中的编码错误和安全漏洞。它涉及将异常测试用例输入到被测试的软件中，试图使其崩溃。美国模糊测试(American Fuzzy Lop，简称 AFL)[28]是最成功的模糊测试工具之一。我们大致将其分为两个主要组件以便于介绍：

执行引擎和模糊测试管理器。前者负责使用目标程序运行测试用例，而后者负责通过基于遗传算法的

变异来生成新的测试用例，协调执行，分析执行信息等。具体地，AFL 首先对目标程序进行仪器化，以便在执行程序时生成和记录分支信息。然后，模糊测试管理器分叉一个新的进程作为执行引擎，使用当前的测试用例运行程序。在执行过程中，仪器化的目标程序消耗测试用例，并将收集的分支覆盖信息记录到本地位图中。模糊测试管理器还将所有本地位图聚合到一个全局位图中，并将新生成的本地位图与全局位图进行比较，以确定是否发现了新的路径。一个能够增加分支覆盖率的测试用例被认为是有趣的，并将在遗传算法中用于计算后续的测试用例。为了报告一个 bug，模糊测试管理器监视目标程序的执行状态，并利用崩溃信息作为指标。

2.2 MCU 固件分析

MCU 是一种专用的片上系统，注重实时处理能力、低功耗和成本。它们被广泛应用于不同的应用领域，如可穿戴设备、智能家居、工业自动化等。MCU 固件的执行环境与传统操作系统显著不同，导致许多现有的二进制分析工具，包括 AFL，在其上无法应用。

与传统软件不同，传统软件假定有一个提供硬件抽象视图的操作系统层，MCU 固件运行在裸金属上，或者仅包含一个简单的多任务管理的操作系统库(例如 RTOS)。因此，它将外设的驱动代码和应用代码编译在一起，形成一个单一地址空间程序。外设的输入/输出操作是通过访问内存映射寄存器来执行的。由于外设的多样性，对 MCU 固件进行动态分析是极具挑战性的。尽管重新主机技术已经取得了一些突破，用于测试固件的与硬件无关的部分[11,13,17,19,22,27,43,47]，但目前尚无现有工作能够测试驱动程序代码。

2.3 硬件支持的指令跟踪收集

程序指令跟踪在许多程序分析应用中非常有帮助，例如性能分析[10,29]、模糊测试[12]、控制流完整性强制执行[18]、根本原因分析[15,16]、调试[32]等。与软件插桩相比，现代处理器支持通过硬件组件捕获指令跟踪，以减少开销。例如，Intel 将其硬件指令跟踪功能称为

Processor Trace 或 PT[21]，从 Broadwell 开始，将其应用到所有的 Core 处理器中。ARM 的对应物称为嵌入式跟踪宏单元(ETM)[1]或程序跟踪宏单元(PTM)[25]。这些实现在很大程度上相似。两者都旨在通过假定相应的机器代码可用来高效地重建整个指令跟踪。具体而言，专用硬件组件会发出一系列控制流数据包。然后，使用解码器将控制流数据与反汇编的机器代码匹配，以重建唯一的执行路径。

2.3.1 追踪数据收集.

ARM MCU 可以选择在芯片上实现用于追踪存储的缓冲区，称为嵌入式跟踪缓冲区(ETB)。然而，根据我们的研究，ETB 在实际芯片上很少被支持。作为替代，ARM 还支持通过物理并行端口将追踪数据流式传输到外部调试器，称为 Cortex Debug+ETM 连接器[3]。这是μAFL 所使用的解决方案。

2.3.2 指令流重构

为了重构执行流程，需要一个解码器来解释追踪数据包，并将其与反汇编指令对齐。控制流数据包携带关于以下信息的信息：a)是否执行了条件分支,b)间接分支的目标,c)异步事件，如异常。

条件分支。ARM ETM 使用 P-header 数据包中的一个位来编码指令的条件是否为真（编码为 1）或假（编码为 0）。真表示相应的指令被执行。这种设计的原因是在 ARM 中，几乎所有指令都可以根据附加到指令的条件标志有条件地执行。以 addeq 指令为例，只有在 Z 标志被设置时才会执行 add 操作。当该指令为分支指令时，如 beq，E 表示进行分支，N 表示不进行分支。

间接分支。间接分支包括间接调用和函数返回。由于间接分支的目标只能在运行时确定，ARM ETM 在发生间接分支时会发出一个包含目标地址的数据包。这些信息被编码到一个分支数据包中。

异步事件。异步异常可能在任何执行点改变控制流。由于当前执行位置已经可以通过 P-header 恢复，因此不需要分支源信息。特别地，分支源由执行的指令长度(由 P-header

确定）与上一个分支目标的基址（由先前的分支数据包确定）相加而得出。ETM 使用现有的分支数据包来编码异步事件，但是通过附加信息来扩展它们。例如，它可以指示此分支是否是由异常而不是正常调用指令引起的。它还指示了如果这是一个异常，则相应的异常编号。此外，ARM MCU 会将现有指令重新用于异常返回。简而言之，如果一条指令导致一组预定义值（EXC_RETURN）的控制流传输，则将其视为从异常返回，并且硬件负责从异常堆栈中获取正确的目标指令指针。ETM 进一步发出一个从异常返回数据包来编码这种事件。通过这种机制，异常入口和返回可以正确配对。

直接分支。有了前面提到的信息，解码器已经可以通过将追踪数据与反汇编指令对齐来恢复整个执行流程。需要注意的是，关于直接分支的追踪信息并不需要，因为直接分支的目标可以通过检查相应的分支指令来确定。然而，ETM 可以选择支持为直接分支发出分支数据包，这样更容易恢复直接分支处的指令流程

2.3.3 追踪数据过滤。

通常来说，随时间收集整个指令追踪是不必要的，因为分析人员可能只对特定的代码区域感兴趣。追踪过滤允许在某些条件下暂停追踪数据的收集。ARMETM 支持基于事件的过滤。它定义了一组 ETM 事件资源，当相应的事件发生时会激活这些资源。这些事件可以由硬件提供的不同比较器进行配置。当匹配发生时，相应的事件变得活动。例如，当指令指针与地址比较器中的值匹配时，相应的事件资源就会被激活。

追踪生成有三种方式进行控制。首先，当一个事件处于活动状态时，它可以直接启用追踪。当它处于非活动状态时，追踪被禁用。其次，可以通过设置一对地址比较器来包含或排除追踪的代码区域。最后，可以通过追踪开始/停止块进行控制。如果发生了一个事件，追踪就会启动。追踪直到该块收到停止信号才会停止，停止信号由另一个事件资源指定。不幸的是，只有最后一种方法被 ARM MCU[1]支持。更糟糕的是，由 DWT 单元提供的比较器资源非常有限。这给我们有效地过滤我们关心的执行追踪数据带来了重大挑战

3. μAFL 设计

在本节中，我们首先概述μAFL 的架构，然后深入探讨两个关键组件的详细设计，即在线追踪收集器和离线追踪分析器，以及它们与 AFL 框架的交互。

3.1 概述

μAFL 是一个针对 MCU 固件设计的新型模糊测试工具，重点关注外设驱动程序代码。其基本理念是继承 AFL 的复杂遗传算法，同时用两个关键组件替换其基于进程的执行引擎，即在线追踪收集器和离线追踪分析器，如图 1 所示。这种设计使得，包括外设驱动程序和闭源库。

正如图 1 所示，主机 PC 和目标板通过调试转接器进行通信，这是嵌入式系统开发中必不可少的工具。首先，主机 PC 通过调试转接器将测试用例传输到目标板上的预留内存中(①)，然后指示目标板开始执行(②)。一旦目标固件达到首次使用测试用例的点，主机 PC 发送命令激活 ETM 功能（③)。然后，在固件执行过程中，生成的指令追踪通过调试转接器同步流式传输到主机 PC(④)。完成一轮执行后，主机 PC 发送另一个命令来停止 ETM(⑤)。收集到的追踪信息然后用于重构执行路径。(⑥)。最终结果被映射到位图中，以确定是否发现了新的路径，并指导按照 AFL 的相同遗传算法生成新的测试用例(⑦)。

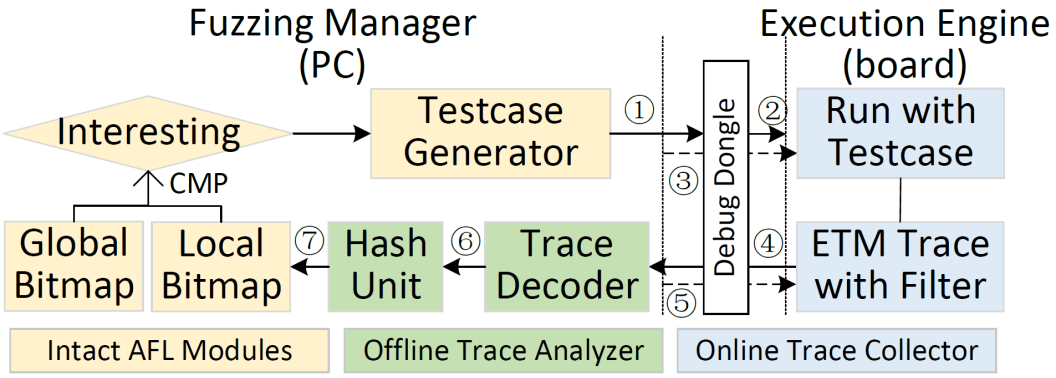

图 1： μAFL 概述

3.2 低级设备控制和模糊测试调度

我们使用 JTAG 或 SWD 接口来对目标设备进行低级控制。通过这些接口，调试转接器可以通过调试访问端口[7]直接访问处理器寄存器和设备内存（包括内存映射的系统配置寄存器）。这也使得我们对目标设备具有最低级别的控制。这是重要的，因为模糊测试通常会导致目标设备进入无响应状态。如果发生这种情况，我们可以通过低级 JTAG/SWD 命令强制进行重置，无需人工干预。

我们还需要将测试用例发送到目标设备。根据生成算法，一个测试用例的大小可能高达数兆字节。我们利用 SEGGER RTT(real time transfer)协议[37]进行高速传输。在底层，RTT 使用 AHB-AP(Advanced High-performance Bus-Access Port)[6]在后台访问内存。它不仅能够提供足够的带宽，还能够实现并行调度。具体而言，我们在后台传输下一个测试用例，同时目标正在执行当前测试用例。此外，我们在 PC 上对先前的测试用例进行分析，与目标执行同时进行。通过在流水线中调度所有任务，μAFL 实现了优化的性能。

3.3 在线追踪收集器

在线追踪收集器负责在固件执行过程中收集 ETM 指令追踪。它需要解决两个主要挑战：1）如何从 PC 向目标板提供测试用例，2）如何选择性地收集感兴趣的代码片段的最小但足够的指令追踪。

3.3.1 测试用例传输

我们预留了两个固定数组来保存测试用例，一个用于当前测试，另一个用于下一个测试，在当前测试过程中后者将在后台传输以提高并行性。如前所述，所使用的通信渠道是 SEGGER RTT。这些数组在一个 noinit 部分声明，以便在初始化过程中不会被 libc 构造函数干扰。数组的大小可以根据需要进行配置。

3.3.2 追踪收集和过滤

要收集执行追踪，我们可以对固件进行仪器化，以便记录每个基本块的转换并将其流式传输到 PC。然而，需要解决两个挑战。a)仪器化需要额外的内存空间和计算资源，资源受限的 MCU 芯片可能无法承受,b)当前的二进制重写技术仍然面临一些基本的技术问题(例如，当前的反汇编器在目标二进制被剥离时无法准确区分引用和字面值)，尤其是当目标二进制被剥离时。

为了解决这个问题，μAFL 利用了 ETM 硬件特性来生成指令追踪。追踪过程对固件是透明的。因此，不需要软件插桩，也不会产生额外的开销。默认情况下，ETM 收集所有的分支信息，这足以恢复测试用例的完整指令追踪。然而，根据我们的实验，这是次优的（见 RQ2 在第 4 节中)。特别地，需要传输和分析大量不相关的数据包。例如，MCU 的启动过程是固定的，不受测试用例的影响。我们可以安全地避免在设备启动期间收集 ETM 数据以节省资源。此外，即使我们有丰富的资源，不相关的数据包也会给模糊器带来噪音，这些噪音无法轻松地去除。例如，一些 MCU 固件是多任务的。收集所有的追踪信息意味着所有任务都被追踪。这带来了不确定性，导致即使测试用例相同，每次运行都会产生不同的追踪。

我们使用 DWT 硬件特性来过滤掉不相关的 ETM 数据包。然而，在 ARM MCU 中，DWT 只实现了有限数量的比较器（在典型实现中为四个)，可以用作过滤器。我们必须优先考虑其使用，以最大程度地减少不相关的数据包。根据第 2.3.3 节中提到的 ETM 触发方法，我们设计了两种在线过滤器，即基于地址的过滤器和基于事件的过滤器。当 MCU 具有更多的比较器资源时，这些过滤器可以组合使用，生成更精细的跟踪数据。

基于地址的过滤器。此过滤器允许分析人员指定要跟踪的连续代码区域。当我们对特定库感兴趣时，这种模式效果最好。此模式需要使用两个比较器来配置区域的起始和结束位置。

基于事件的过滤器。μAFL 还支持基于事件的过滤器，其中某些事件触发 ETM 的开关。该事件可以是执行特定内存范围内的指令，或者从/向特定地址读取/写入特定值。两者都需要使用两个比较器。我们将前者称为指令触发器，后者称为数据触发器。

指令触发器在跳过设备引导过程中非常有用。我们使用清单 1 中的代码片段作为示例。第 4-7 行是设备引导的一部分，与测试用例无关。第 12 行是固件的主要逻辑，放在一个无限循环中。这是 MCU 编程中的范例 - 主要操作在循环中被持续执行，以感知环境数据并相应地进行处理。我们通过添加三行代码（第 9、13、14 行）来控制代码。需要注意的是，虽然μAFL 不需要目标库的源代码，但需要调用目标库的源代码来使模糊测试过程变得简单。fuzz_stop 是一个标记，表示模糊测试是否应该停止。当测试用例已经用完时，它可以由固件异步地更改，或者由调试工具异步地更改。通过配置第 9 行的指令地址来启动 ETM，配置第 14 行的指令地址来停止 ETM，μAFL 可以有效地聚焦于固件的主要逻辑部分。

```
1.  int fuzz_stop = 0;
2.  int main(void)
3.  {
4.      MPU_Config();
5.      SCB_EnableICache();
6.      SCB_EnableDCache();
7.      HAL_Init();
8.      // ...
9.      fuzz_stop = 0;
10.     while (1)
11.     {
12.         MX_USB_HOST_Process();
13.         if (fuzz_stop)
14.             break;
15.     }
16. }
```

Listing 1: 包含初始化代码、主要应用程序逻辑和μAFL 引擎的代码片段

数据触发器提供了细粒度的跟踪能力，我们利用这一特性来跟踪指定的任务。更具体地说，在多任务环境中，我们使用数据触发器模式来过滤掉其他任务和操作系统内核（如中断处理程序和调度），这被视为对模糊测试器的噪音。我们观察到，在实时操作系统环境

中，每个任务都有其任务控制块（TCB）在固定的位置，而不受所使用的测试用例的影响。因此，我们可以配置 DWT，在全局指针（在 FreeRTOS 中为 pxCurrentTCB）写入当前任务的 TCB 地址时，打开 ETM。当向该指针写入任何其他值时，表示目标任务已被切换，ETM 被关闭。

3.4 离线跟踪分析器

对于每个测试用例的执行，离线跟踪分析器会处理来自目标设备的 ETM 数据包。它首先在不解码原始 ETM 数据的情况下恢复分支信息。这是通过一种特殊的基本块实现的。然后，从基本块转换中导出的分支信息用于更新由 AFL 维护的代码覆盖率位图。所有其他 AFL 组件保持不变，包括新的路径识别模块、用于生成新测试用例的遗传算法等。

3.4.1 在不解码 ETM 的情况下计算分支信息

为了捕获分支信息，AFL（在 QEMU 模式下）动态捕获基本块的转换，并使用基本块的地址来填充位图，以下是相应的代码[28]。

```
1.  cur_location =( block_address > >4) ^( block_address < <8)
2.  ;
3.  shared_mem [ cur_location ^ prev_location ]++;
4.  prev_location = cur_location > >
```

Listing 2:AFL 在 QEMU 模式下的分支覆盖率计算

在 Listing2 中，shared_mem 指的是当前测试用例的 本地位图，block_address 是当前基本块的地址。在第 1 行，block_address 被输入到一个简单的哈希函数中，以获取当前基本块的随机标识，表示为 cur_location。对于图 2 中显示的路径$A \rightarrow B \rightarrow C$，基本块转换序列为 $0x8000546 \rightarrow 0x800054E \rightarrow 0x8000584$，而对于路径$A \rightarrow C$，基本块转换序列为$0x8000546 \rightarrow 0x8000584$。根据上面显示的代码，这两条路径生成了两个不同的位图，AFL 可以利用这些位图来找到有趣的测试用例。

利用 ETM 跟踪，我们可以恢复相同的分支覆盖信息。具体地说，通过沿着反汇编指令序列进行遍历，并将其与解码的 ETM 数据包对齐，我们可以恢复整个指令跟踪，并进一步

重建与 AFL 相同的分支信息。然而，根据文献[18]的说法，这会产生非常大的开销，并且我们的实验也验证了这一点（见第 4.1 节）。

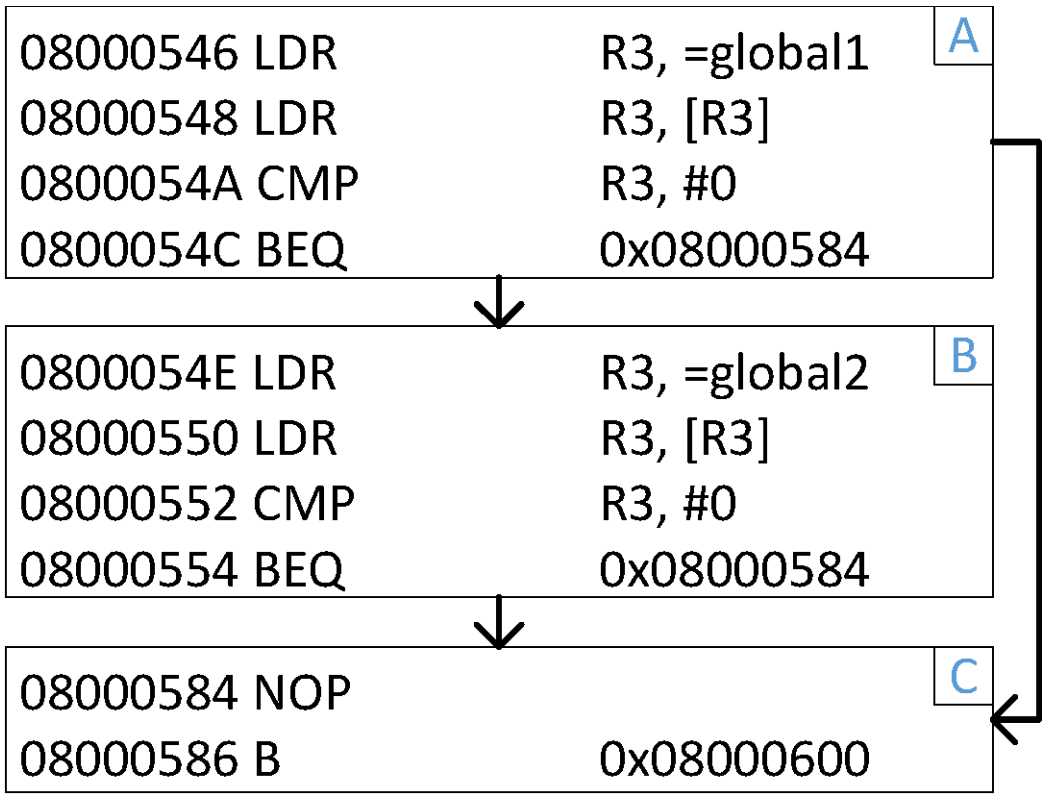

图 2:某个示例代码的控制流图

为了避免昂贵的代码反汇编和 ETM 解码，我们提出了一种新颖的机制，直接使用原始的 ETM 数据包来捕获分支覆盖信息。一个直观的想法是将每个 ETM 分支数据包的目标地址作为每个基本块的起始地址。然而，它无法区分图 2 中的两条路径，因为地址 $0x800054C$ 和 $0x8000554$ 处的条件分支无论分支是否被执行都不会生成任何分支数据包。(回想一下，P-header 用于记录分支是否被执行的信息)。幸运的是，我们在 ETM 手册中找到了一种方法[1]，可以通过设置 ETMCR 寄存器的第八位来覆盖这种行为。具体来说，启用此选项后，ETM 将会为实际被执行的直接分支生成分支数据包。因此，路径$A \rightarrow B \rightarrow C$将会产生一系列的 ETM 数据包（$0x8000546, EEEN, EEEE, 0x8000584$），而路径$A \rightarrow C$则会产生一系列的 ETM 数据包（$0x8000546, EEEE, 0x8000584$）。

这里，E是 P-header 中的一个位，表示指令的条件为真，而N表示相反情况，如第 2.3.2 节所述。可以看到，后者路径直接从基本块A跳转到C，因为地址 $0x800054C$处的分支条件为真。因此，在E位后面生成了一个目标为 $0x8000584$ 的分支数据包。相反地，对于前者路径，地址 $0x800054C$处的分支条件为假，因此在该处不会生成分支数据包。但是，地址 $0x8000554$ 处的分支条件为真，这导致生成一个目标为 $0x8000584$ 的分支数据包。通过比较两个ETM跟踪结果，很明显我们可以区分这两条路径，因为两个分支目标之间的P-header位是不同的。

为了解释μAFL 捕获的分支覆盖信息，我们首先解释一种特殊的基本块，它是通过线性代码序列和跳转（LCSAJ）分析生成的[44]。我们称之为 LCSAJ_BB。一个 LCSAJ_BB 是一个指令序列，从上一个被执行的分支目标开始，到下一个实际被执行的分支指令结束。根据这个定义，在路径$A \rightarrow C$中，基本块 A 本身就是一个 LCSAJ_BB，而在路径$A \rightarrow B \rightarrow C$中，基本块 A 连接上基本块 B 构成一个 LCSAJ_BB，因为A结尾处的分支没有被执行。在μAFL 中，我们通过结合上一个分支目标获取的基地址和下一个 LCSAJ_BB 前的 P-header 位流来表示一个 LCSAJ_BB，形式上表示为$(BB_base, BB_bitstream)$。例如，在路径$A \rightarrow C$中，A的 LCSAJ_BB 被编码为 $(0x8000546, 1111)$，而在路径 $A \rightarrow B \rightarrow C$ 中，$A|B$ 的 LCSAJ_BB 被编码为 $(0x8000546, 11101111)$。请注意，可以在不参考汇编代码的情况下获得这些信息。在第 3.4.2 节中，我们将解释如何使用 LCSAJ_BB 转换来计算分支覆盖，以便与 AFL 进行对接。值得注意的是，我们的方法并不会生成与 AFL 相同的位图。但是，它实现了相同的路径敏感性，因为基本块转换的任何变化都将反映在相应的 LCSAJ_BB（BB_base或$BB_bitstream$）上的变化。

非确定性。尽管在线跟踪收集器已经可以过滤掉大量相关的 ETM 数据包，但受到硬件的限制，仍然会发出许多嘈杂的数据包。例如，没有机制来抑制异常处理程序的跟踪，而异常处理程序的发生是非确定性的。这会给模糊器增加不稳定性，因为相同的测试用例在不同的运行中会生成不同的执行跟踪。为了解决这个问题，μAFL 还提供了一个离线异常过滤器。具体来说，我们利用嵌入在 ETM 分支数据包中的异常信息来确定异常的进入点和退

出点。分析人员可以选择是否丢弃处理程序执行期间生成的 ETM 跟踪。同样，在此过程中不需要进行反汇编操作。

3.4.2 将分支信息映射到位图

在本节中，我们解释了如何将基于 LCSAJ_BB 的分支信息映射到 AFL 维护的位图中。按照列表 2 中显示的 AFL 设计，我们的目标是将一个由 $(BB\_base, BB\_bitstream)$ 表示的 LCSAJ_BB 转换为一个唯一的数字，该数字将用于取代列表 2 中的 cur_location（基本块的唯一标识）。为了充分扩散每个 LCSAJ_BB 中包含的信息，我们采用基于 MurMurHash[8]的轻量级哈希算法。输出是一个随机整数BB_ID。正如前面提到的，整个算法替换了列表 2 中的第 1 行。我们将位流分成 5 位的块，并对它们进行位异或运算，得到一个从 0 到 31 的数字t(FoldAndXor())。然后，t用于与BB_base进行混合和移位。结果进一步分成两部分，并与一些魔术数进行混合。虽然这种设计是临时的，但根据我们的评估，它有效地随机化了编码的 LCSAJ_BBs，使得生成的BB_ID不会在 AFL 位图中产生太多的冲突。

Algorithm 1: 用于将一个 *LCSAJ_BB* 转换为一个随机整 ID 的哈希函数

Input: *(base, bitstream)* ← (*BB_base*, BB_bitstream)
MAP_SIZE ← length of bitmap in bytes
Output: *BB_ID*
Function HASH(*base*, *bitstream*):
$t \leftarrow FoldAndXOR(bitstream)$;
$base \leftarrow base + t$;
$left \leftarrow (base \ll (32-t)) \mid (base \gg t)$;
$right \leftarrow (base \ll t) \mid (base \gg (32-t))$;
$BB\_ID \leftarrow (left \mid right)$;
$BB\_ID \leftarrow (BB\_ID \oplus (BB\_ID \gg 16))$;
$BB\_ID \leftarrow (BB\_ID * 0x85ebca6b)$;
$BB\_ID \leftarrow (BB\_ID \oplus (BB\_ID \gg 13))$;
$BB\_ID \leftarrow (BB\_ID * 0xc2b2ae35)$;
$BB\_ID \leftarrow (BB\_ID \oplus (BB\_ID \gg 16))$;
$BB\_ID \leftarrow ((BB\_ID \gg 4) \oplus (BB\_ID \ll 8))$;
$BB\_ID \leftarrow (BB\_ID \wedge (MAP\_SIZE - 1))$;
return *BB_ID*;
End Function

3.5 崩溃/挂起检测

由于缺乏内存保护机制，嵌入式系统中的崩溃检测是具有挑战性的。

μAFL 依赖内置的异常处理机制来检测异常的固件行为。具体来说，我们使用矢量捕获特性[5]来标记关注的异常，例如 Hard Fault、Mem Manage、Bus Fault、Usage Fault 等。这些异常指示了关键的系统错误，因此可以用作崩溃信号。通过矢量捕获，当此类异常发

生时，芯片不会陷入相应的处理程序，而是进入调试状态，这可以被调试接口自动捕获。然后，μAFL 进一步检查故障状态寄存器和故障地址寄存器以检查根本原因。为了通知模糊测试管理器一个成功的执行，在测试代码的末尾，我们放置一个带有魔术数字作为参数的 BKPT 指令。如果当前测试正确终止，执行将在此 BKPT 指令处进入调试状态。我们进一步检查参数的值以确认成功的执行。最后，如果在指定的时间内（我们使用两秒作为经验值）模糊测试管理器没有捕获任何调试状态，那么将标记为挂起。

4. 实现与评估

我们在 AFL2.56[46]的基础上实现了μAFL 的原型，在 PC 端增加了约 2000 行 C 代码。我们使用 SEGGER J-Trace Pro 调试接口[39]控制主机 PC 和目标 ARM Cortex-M 评估板之间的通信。控制逻辑是使用 SEG-GER 提供的 SDK 在 PC 端实现的[38]。在图 3 中，我们展示了使用我们的原型模糊测试 NXP TWR-K64F120M 评估板上的固件样本的设置。关键组件与图 1 中的架构图相对应。

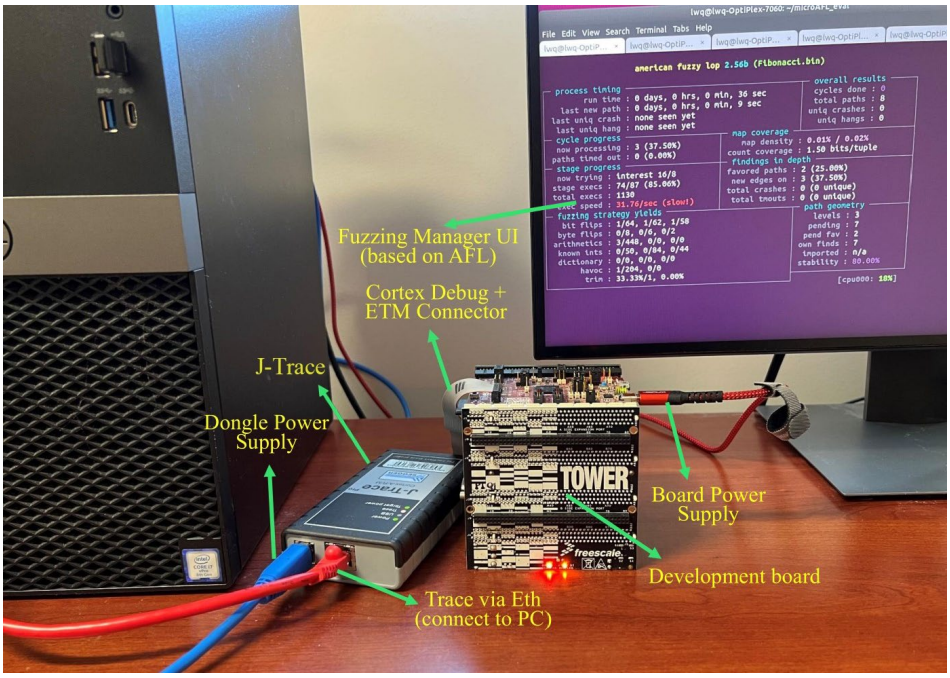

图 3：对固件样本进行模糊测试的 μAFL 是在 NXP TWR-K64F120M 板上进行实验

我们评估了我们的μAFL 原型的性能，以回答五个研究问题。答案预期展示了μAFL 的独特优势以及作为嵌入式固件测试新解决方案的有效性。

RQ1:直接处理原始 ETM 追踪的基于 LCSAJ_BB 的方法是否比完全恢复指令流的方法提高了性能？

RQ2:在线过滤器是否有助于提高μAFL 的性能？

RQ3:μAFL 对单轮模糊测试的（每个子过程的）开销有多大？

RQ4:μAFL 的总体性能如何，与现有工作相比如何？

RQ5:μAFL 在定位实际固件外围驱动程序中的错误方面有多有效？

实验设置。运行模糊测试管理器的个人电脑配备有 Intel Core i7-8700 CPU@3.2GHz、8 GB DDR4 RAM 和 SSD 存储。我们使用 NXP TWR-K64F120M 评估板作为执行引擎，并使用相应的 SDK 对从 RQ1 到 RQ4 的性能实验进行了评估。为了回答 RQ5，我们提供了关于在 NXP TWR-K64F120M 和 STM32H7B3I-EVAL 评估板上进行长期模糊测试并针对其相应的 SDK 向各种项目发现的实际错误的详细案例研究，这两个板子都有可用的 ETM 引脚。

用于评估的固件样本。对于 RQ1 到 RQ4，我们使用 NXP SDK 中提供的样本代码。首先，样本 Fibonacci 实现了一个递归函数，计算 Fibonacci(1,000,000)。此样本不涉及任何外设，作为我们评估的基线。其次，样本 I2C 涉及使用简单的外设 I2C。它只是通过 I2C 总线与个人电脑通信。第三，样本 UART 涉及 UART 的使用，也与个人电脑通信。第四，样本 USB 将评估板用作 USB 主机，访问格式为 FAT 文件系统的 USB 磁盘。第五，样本 SDCard 识别并初始化插入到板载 SD 卡槽中的微型 SD 卡。第六，样本 Enet 使用以太网通过 IPv4 与个人电脑通信。最后，样本 MMCAU 使用硬件密码加速单元(CAU)[33]完成加密操作，如 AES、DES3、SHA 等。注意 CAU 库是闭源的。我们在列表 2 中列出了包括大小和基本块数量在内的固件信息。对于 RQ5，我们使用 NXP SDK 和 STM32 SDK 中提供的样本代码。虽然我们在两个 SDK 中测试了多个驱动程序，但我们特别选择了 USB 作为案例研究，以展示 μAFL 在发现复杂外设中的错误方面的能力。所有样本的应用程序级逻辑非常简单，因为我们不试图在高级代码中找到错误。相反，我们确保样本中包含了核心外设功能，目标是向低级驱动程序代码提供异常输入，以触发错误。

4.1 完全恢复指令流的开销

正如在第 3.4.1 节中讨论的，μAFL 通过避免对固件进行反汇编并对齐 ETM 跟踪以恢复完整的指令流来降低处理开销。在本节中，我们将演示使用基于 LCSAJ_BB 的方法如何减少 PC 上的性能开销。在基线方法的实现中（完全反汇编固件并解码 ETM 数据），我们使用了流行的反汇编框架 Capstone[31]。我们通过将收集到的 ETM 数据与反汇编的指令进行对齐来跟踪固件的执行。每当遇到新的基本块时，我们一次性对整个基本块进行反汇编，这也被缓存供将来使用。在恢复指令跟踪后，我们遵循列表 2 中的步骤来填充位图。值得一提的是，我们采用与μAFL 相同的过滤策略，以确保公平比较。我们测量了每个样本在一小时内的总执行次数。结果显示在表 2 的第 4 列中。与μAFL 的性能相比（第 6 列），我们确认μAFL 通常比基于反汇编的方法要快得多。我们观察到平均改进了 1.03 倍至 4.81 倍

4.2 滤波器性能

为了回答 RQ2，我们禁用了提出的在线滤波器，并比较了。与之前一样，我们记录了一小时内的执行次数。结果分别显示在表 2 的第 6 列和第 5 列中。首先，我们可以看到在线滤波器在我们测试的所有样本中通常都提高了。对于非常大的样本（例如 Enet、USB 和 SD 卡），滤波器轻微提高了。然而，对于较不复杂的样本，如 MMCAU，改进非常显著。这是因为没有滤波器，ETM 需要收集从 ResetISR 到每次运行结束的整个指令跟踪，这不仅增加了调试硬件传输原始跟踪数据的负担，还需要更多时间让离线解码器将其解码并映射到位图中。对于较大的样本，由于每次运行的执行时间较长，这种开销可以在整个执行过程中摊销。

Sample	BB#	Size (bytes)	Dis&Dec*	*μAFL* w/o Filter	*μAFL*
Fibonacci	9,927	12,064	100,830	33,386	104,394
I2C	13,343	14,888	112,666	27,732	116,487
UART	9,899	12,856	37,003	21,087	50,571
USB	36,431	40,024	465	2,211	2,236
SD Card	21,104	25,328	592	2,207	2,335
Enet	14,475	18,504	685	1,089	1,116
MMCAU	12,186	17,712	58,155	26,393	74,830

表 2:不同设置下的固件大小信息和模糊测试性能（每小时执行次数）

4.3 开销细分

每个模糊测试轮包括五个子进程。(1)Reset2Start 从设备引导（即 Reset_Handler 开始）到模糊测试开始点（即开始读取测试用例）的时间。请注意，我们在此子进程中未收集分支覆盖信息，因为它在所有测试用例中保持不变。(2)StartTrace 衡量了准备测试所花费的时间，包括从 PC 传输测试用例到板上以及配置 ETM 和 DWT 功能。(3)Start2End 是从模糊测试开始点到结束点所花费的时间。在此子进程中收集分支覆盖信息。(4)StopTrace 是在禁用 ETM 跟踪并等待 ETM 跟踪传输完成时花费的时间。(5)Analysis 是用于分析 ETM 数据包的时间。

在这个实验中，我们测量了每个子进程的执行时间，以评估其对总体开销的影响。我们对每个样本进行了 1,000 轮执行，计算了平均执行时间。图 4 展示了执行时间的细分。我们可以看到，在所有样本中，Reset2Start 和 StartTrace 子进程的执行时间保持稳定，而其他子进程的执行时间随着固件变得更加复杂而增加。这是因为在更复杂

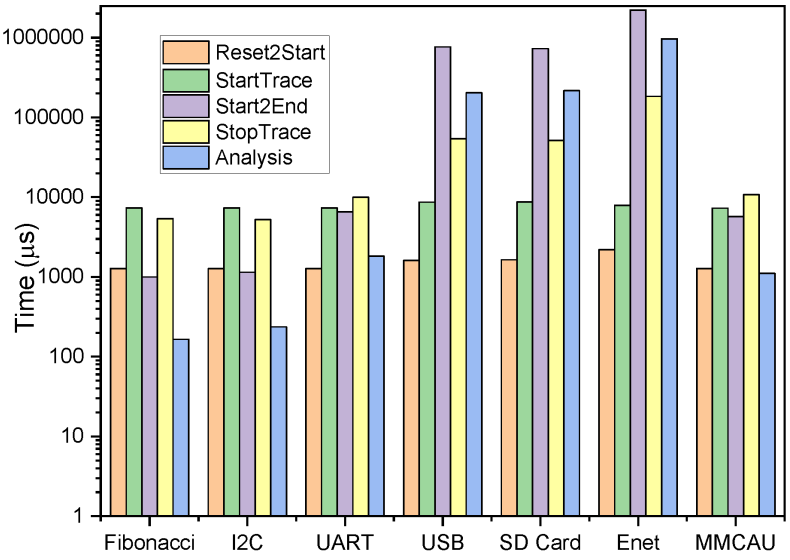

图 4：*μ*AFL 的开销细分（y 轴为对数刻度）

的固件样本中生成了更多的 ETM 数据包。这不仅会增加由于等待跟踪传输完成而产生的 StopTrace 操作的开销，还会增加由于解码和哈希计算增加而产生的 Analysis 操作的开销。我们还观察到对于简单的固件来说，执行速度非常快。对于它们来说，开销主要由 SEGGERSDK 中提供的黑盒函数 Start-Trace 和 StopTrace 子进程所主导。随着固件变得更加复杂，完成 Start2End 子进程和分析大量指令跟踪所需的时间变长。

4.4 μAFL 整体性能

我们评估了，以回答 RQ4。我们对每个固件运行了大约两天，并记录了执行次数、总执行时间、覆盖路径数量以及崩溃和挂起的数量，如表 3 所示。我们还要指出，当生成的跟踪数据非常大（例如，超过 200MB）时，调试硬件可能变得不够可靠。调试硬件内部的缓冲区可能会耗尽，导致溢出和跟踪数据丢失。如果发生这种情况，我们必须强制终止执行并重置故障周期。这导致执行速率较低，因为我们必须放弃有问题的执行。在极少数情况下，调试硬件中的错误可能无法检测到。如果发生这种情况，我们还观察到一些误报。这解释了表中的误报。当我们重放在模糊测试期间触发崩溃/挂起的相同测试用例时，结果无法复现。

Project	Time(s)	Executions	Exec/sec	Paths	Crashes/Hangs
I2C	172,893	5,340,552	30.8893	2	0/0
UART	172,838	1,947,980	11.2706	27	1/0
USB	172,803	148,398	0.8588	201	0/500
SD Card	171,693	149,063	0.8682	53	0/0
Enet	174,675	52,961	0.3032	84	14/0
MMCAU	173,180	3,268,671	18.8744	95	0/0

表 3:Fuzzing 性能表现

与现有工作的比较。正如在第 1 节中解释的那样,。与之相关的工作包括 Avatar[45]，它在 QEMU 中模拟固件，但将外设操作转发到实际的开发板，并且纯模拟的解决方案，如 P2IM[17]和μEmu[47]。原始的 Avatar 只利用实际的硬件通过固件初始化阶段，然后使用符号

执行来分析从不访问外设的代码。后来，Avatar 被大幅重构，用于多目标编排，在 Avatar2[30]中实现了这一目标。然而，在这两者上默认不支持模糊测试。虽然基于模拟的方法具有很好的可扩展性，但不能保证对驱动程序代码进行模糊测试时的足够真实性。例如，具有复杂外设的固件可能无法启动。

在本节中，我们忽略了模糊测试的有效性，而是关注μAFL、Avatar2、P2IM 和μEmu 所实现的原始模糊测试速度。我们选择了 Avatar2 进行实验，因为它在开发中，并且具有更友好的 API 设计。我们使用了两个样本。除了之前提到的 Fibonacci 样本之外，我们还评估了一个 Console 固件样本，该样本也在 P2IM[17]和μEmu[47]中使用。它通过 UART 外设提供一个简单的交互式 Shell。值得注意的是，我们尝试使用具有更复杂外设（例如 USB 或 Ethernet）的固件，但是没有相关的工作支持它们。实际上，Avatar2 不支持 DMA，

而这对于复杂外设是必不可少的。P2IM 和μEmu 无法模拟 USB 或 Ethernet，因此无法启动固件。我们增强了原始的 Avatar2 框架，使其具有模糊测试能力，并提高了 JLink（SEGGER 的调试解决方案的名称）目标的稳定性，以管理开发板并协调模糊测试过程 2，与我们在。固件在 QEMU 中开始执行，并在需要时将 I/O 请求转发到开发板。对于 P2IM 和μEmu，我们直接使用原始论文中指定的源代码构建系统。但是，在对 Fibonacci 进行模糊测试时，我们进行了一些调整。具体来说，P2IM 和μEmu 仅在测试用例中的所有字节都被用完时才标记执行的结束。这导致在对 Fibonacci 进行模糊测试时不断超时。为了解决这个问题，我们在源代码中手动插入了许 getchar()调用，以模拟使用测试用例。

结果总结在表 4 中。基于模拟的解决方案由于更高的计算能力，因此在很大程度上优于硬件在环解决方案。更重要的是，它们避免了 PC 和板之间耗时的同步。另一方面，2，因为它在同步方面的开销较小。对于计算密集型任务（Fibonacci），差距缩小了，因为它们不经常访问外设，从而避免了同步。

Sample	**P²IM**	**μEmu**	**Avatar²**	***μAFL***
Console	139,860	29,513	1,766	8,261
Fibonacci	209,478	904,617	2,623	8,670

表 4:与相关工作的比较（每小时执行次数）

4.5 现实世界的固件模糊测试

为了展示，我们对多个具有复杂外设驱动程序的样本进行了为期两天的模糊测试，包括以太网、USB 和 SD 卡。在每个样本上运行模糊测试器两天后，我们在 STM32 SDK 发布的 USB 驱动程序中发现了十个先前未知的漏洞，以及在 NXP SDK 中发现了三个。接下来，我们以 USB 为案例研究，展示。

我们的模糊测试器专注于 USB 枚举过程，在该过程中 USB 设备被主机识别。在我们的评估中，我们将 MCU 板用作 USB 主机，并将 SanDisk USB 磁盘用作 USB 设备。我们假设 USB 磁盘是恶意的，并可以向 MCU 板发送任意数据。我们手动识别了从 USB 磁盘接收数据的程序点，并用 AFL 生成的测试用例替换它们。

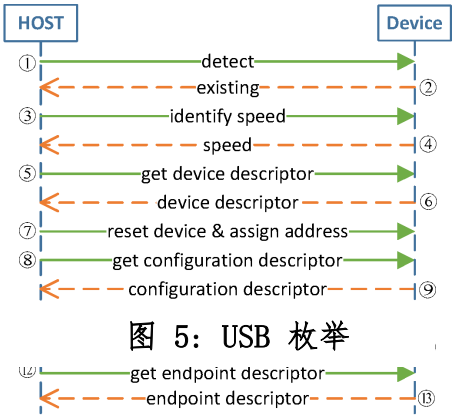

图 5：USB 枚举

如图 5 所示，USB 枚举非常复杂，涉及多轮交互。最重要的是，USB 设备需要发送多个描述符来指定设备的属性。当 USB 主机解析这些描述符时，如果相关字段未经适当清理，可能会发生内存错误。在为期两天的模糊测试活动中，我们发现了 13 个以前未公开的漏洞。所有这些漏洞都已报告给供应商并修补。请注意，这些漏洞影响使用 STM32SDK 或 NXPSDK 的所有 MCU 设备。我们将找到的漏洞分类如下。

描述符中未经清理的长度属性。很多漏洞是由于在不同描述符的长度字段未经适当清理而引起的。例如，设备描述符的属性 bMaxPacketSize(⑥)定义了从主机发送到设备的数据包的允许最大大小。如果这个值被恶意指定并且没有经过检查，MCU 主机中的驱动程序将错误地配置一个有问题的管道，并分配一个意外大小的缓冲区，导致拒绝服务。我们在解析配置描述符(⑨)、接口描述符(⑪)和端点描述符(⑬)的字段 wTo-talLength，以及端点描述符(⑬)的字段 wMaxPacket-Size 中发现了类似的漏洞。这些漏洞的后果包

括缓冲区溢出和拒绝服务。我们工具在这个类别中发现的相关 CVE 包括 CVE-2021-34259、CVE-2021-34260、CVE-2021-34268、CVE-2021-38258 和 CVE-2021-38260。

未经检查的硬件支持检查。配置描述符中的字段 bmAt-tributes 指示配置的不同功耗参数。当相应位被设置时，USB 驱动程序尝试激活远程唤醒功能以节省电源。但是，驱动程序未检查设备是否实现了此功能，导致由于请求失败而系统挂起。CVE-2021-34261 属于此类。

不合逻辑的端点地址。端点描述符中的字段 bEndpointAd-dress 指定了端点的标识。主机通常应为数据接收和传输分别建立 IN 端点和 OUT 端点。驱动程序默认接受这一点，并且不检查端点描述符的内容。当一个畸形的端点描述符指定了相同的方向时，主机可能会丢失 IN 管道或 OUT 管道，导致系统挂起。CVE-2021-34267 属于此类。

未经检查的轮询间隔。端点描述符中的字段 bInterval 指示轮询端点数据传输的间隔。它将用于为个别 USB 应用指定不同的轮询间隔，例如音频流。当此值为负数时，轮询逻辑将挂起设备。CVE-2021-34262 属于此类。

5. 讨论

模糊测试驱动程序代码。与模糊测试常规库不同，对外设驱动程序进行模糊测试涉及与外部物理世界的交互，这在模糊测试中引入了许多非确定性。此外，与通用库相比，外设驱动程序通常需要通过与外设的内存映射寄存器进行读取或写入来操作，这也增加了模糊测试中的非确定性。此外，外设的行为通常被建模为状态机，其状态转换由许多事件触发，例如中断和 MMIO 交互。因此，有效地对外设驱动程序进行模糊测试需要深厚的领域知识，正如我们在对 USB 枚举实现进行模糊测试中所演示的那样。由于这一点，用于模糊测试外设驱动程序的代码测试通常是逐案进行的。作为一项一般性指导方针，我们应该找到驱动程序与硬件外设进行交互的程序点。这些是从不受信任的源接收输入的地方，因此应该被约束以读取测试用例。我们计划在未来对更复杂的驱动程序进行模糊测试。

适用性。我们的方法依赖于 ETM 硬件特性，因此无法用于没有此特性的芯片。幸运的是，根据我们的经验，ETM 在 MCU 芯片中非常流行。真正的问题是，由于 PCB 设计的额外成本，只有很少一部分开发板具有与 ETM 进行接口的物理引脚布局。我们认为这不影响我们的工具在内部测试或模糊测试供应商 SDK 时的适用性。首先，开发人员可以轻松地组装一个带有 ETM 引脚布局的 PCB 板[41]。其次，模糊测试 SDK 的结果通常适用于同一芯片供应商的其他 SDK。具体来说，芯片供应商倾向于为类似的产品线维护一组通用的设备驱动程序。如果单个开发板具有 ETM 引脚布局，则发现的漏洞可能适用于其他芯片。以 STM32MCU 为例，最初，我们的工具仅为 STM32H7 系列芯片发现了 CVE-2021-34268，因为我们只能访问支持 ETM 的 STM32H7B3I-EVAL 板。然而，该漏洞影响了 STM32 芯片的几乎所有产品线，包括 STM32F4[3]、STM32F1[4]等

6. 相关工作

6.1 MCU 固件模糊测试

现有的 MCU 固件模糊测试方法可以分为五类，如 Li 等人的工作[43]所示。

模拟是最直观的方法。然而，完全模拟市场上各种 MCU 是不可能的。现有的基于 QEMU 的解决方案，例如 P2IM[17]、DICE[27]、Laelaps[11]、PRETENDER[19]、μEmu[47]和 Jetset[22]，通过利用各种方法（如机器学习、符号执行等）来近似外设的行为，以成功运行固件，进行了一种折衷。HALucinator[13]通过抽象和替换硬件接口，而不执行任何实际的外设功能，进一步简化了这个过程。所有这些方法都展示了良好的性能和成本效益。与 HALucinator 一样，para-rehosting[43]通过在通用硬件上抽象 MCU，以便从 AFL[46]和 Address-Sanitizer[40]等现成的测试套件中受益，提供了一个统一的平台，具有最强大的能力。此外，它可以极大地提高模糊测试的性能。然而，这两种技术都无法深入到驱动程序层。尽管外设转发机制（例如 Avatar[45]）与硬件进行交互，但主要用于支持上层的模糊测试。此外，仅使用转发外设

操作将导致上下文丢失问题。此外，模拟器和实际设备之间的完整固件同步会带来额外的开销。

6.2 外设漏洞检测

USBFuzz[36]提出了一个框架，通过在虚拟化内核中使用模拟USB设备，向通用操作系统USB驱动程序应用模糊测试。模拟USB设备提供测试用例并打破硬件/软件界限。然而，这种方法无法用于 MCU 固件模糊测试，因为 a)与通用操作系统主机的内核不同，由于缺乏MMU，MCU无法运行任何现有的模糊测试器，b)没有适用于MCU的开箱即用的设备模拟器，且c)MCU外设繁多且异构。因此，仍然需要单独考虑MCU外设。

Facedancer[23]是一个早期的尝试，用于对MCUUSB驱动程序进行模糊测试，提出使用一个板子，该板子以各种描述符响应并伪装成不同的USB设备。作为模糊测试器和目标MCU之间的中间人，该板子有助于提供测试用例并检查操作状态。然而，这种虚拟模糊测试方法在大多数情况下并不有效，因为它无法从目标获取任何反馈以生成位图并引导测试用例生成。此外，该板子仅支持USB驱动程序。

FIRMUSB[20]采用了一种领域特定的方法来检测USB漏洞。为了识别不合规的行为，它将从已知USB数据库中生成的模型与符号执行检索的模型进行对比。因此，FIRMUSB可以在不在真实板上运行二进制固件的情况下检测到固件的漏洞。然而，与FaceDance类似，这一工作仅专注于测试USB固件

7. 总结

我们提出了μAFL，一种用于 MCU 固件的非侵入式反馈驱动模糊测试平台，特别针对底层外设驱动程序。μAFL将执行引擎与原始的AFL框架解耦，并使用开发板执行测试用例。为了使执行引擎与AFL的其余部分之间能够进行通信，我们利用嵌入式系统开发环境中普遍可用的调试dongle。为了有效地获取指令跟踪，我们依赖于ETM和DWT硬件特性。最后，我们提出使用动态基本块来减少解码和分析ETM数据的开销。我们已经针对两个流行的MCU

供应商的 SDK（NXP 和 STMicroelectronics）对我们的原型实现进行了评估。该原型帮助我们在供应商 SDK 中的 USB 堆栈中发现了 13 个零日漏洞，并分配了八个 CVE。

四、外文原文

μAFL: Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware

Wenqiang Li
State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
School of Cyber Security, UCAS
Beijing, China
liwenqiang@iie.ac.cn

Jiameng Shi
Department of Computer Science, the University of Georgia
Athens, Georgia, USA
jiameng@uga.edu

Fengjun Li
Department of Electrical Engineering and Computer Science,
the University of Kansas
Lawrence, Kansas, USA
fli@ku.edu

Jingqiang Lin
School of Cyber Security, University of Science and Technology of China
Hefei, Anhui, China
linjq@ustc.edu.cn

Wei Wang
State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
wangwei@iie.ac.cn

Le Guan
Department of Computer Science, the University of Georgia
Athens, Georgia, USA
leguan@uga.edu

ABSTRACT

Fuzzing is one of the most effective approaches to finding software flaws. However, applying it to microcontroller firmware incurs many challenges. For example, rehosting-based solutions cannot accurately model peripheral behaviors and thus cannot be used to fuzz the corresponding driver code. In this work, we present *μAFL*, a hardware-in-the-loop approach to fuzzing microcontroller firmware. It leverages debugging tools in existing embedded system development to construct an AFL-compatible fuzzing framework. Specifically, we use the debug dongle to bridge the fuzzing environment on the PC and the target firmware on the microcontroller device. To collect code coverage information without costly code instrumentation, *μAFL* relies on the ARM ETM hardware debugging feature, which transparently collects the instruction trace and streams the results to the PC. However, the raw ETM data is obscure and needs enormous computing resources to recover the actual instruction flow. We therefore propose an alternative representation of code coverage, which retains the same path sensitivity as the originalAFLalgorithm, but can directly work on the raw ETM data without matching them with disassembled instructions. To further reduce the workload, we use the DWT hardware feature to selectively collect runtime information of interest. We evaluated *μAFL* on two real evaluation boards from two major vendors: NXP and STMicroelectronics. With our prototype, we discovered ten zero-day bugs in the driver code shipped with the SDK of STMicroelectronics and three zero-day bugs in the SDK of NXP. Eight CVEs have been allocated for them. Considering the wide adoption of vendor SDKs in real products, our results are alarming.

CCS CONCEPTS
• Security and privacy → Embedded systems security

KEYWORDS
• firmware security, fuzzing, microcontroller, IoT, ETM

1. INTRODUCTION

Internet-of-Things (IoT) has become an integral part of our digital lives. For example, many people install smart thermostats to remotely control the temperature and humidity of their homes. Smart alarm systems are used to monitor home and workplace and raise alarms when detecting burglars. Fitness trackers and smart health bands are also widely used to continuously monitor personal health data such as heart rate and blood oxygen level. The key component of many IoT devices is the microcontroller unit (MCU), which is a tiny, custom-built, and cost-efficient system-on-chip (SoC) [26].

The rapid evolution of the MCU ecosystem, on the one hand, has made our lives easier and more convenient than ever before,

on the other hand, it also introduces a large number of vulnerable MCU products in the wild. For example, several high-profile vulnerabilities have been reported recently for the ESP8266 and ESP32 communication and WiFi co-processors, which have been adopted in millions of IoT devices. These vulnerabilities (e.g., *EAP client crash*, *zero PMK installation* and *beacon frame crash* [35]) allow the adversaries to hijack or crash the session of ESP32/ESP8266 products. Meanwhile, vulnerabilities of the FreeRTOS TCP/IP stack can be used to launch remote code execution and steal private information [48, 49]. More recently, BadAlloc [9] leaves a large number of IoT devices exposed to adversaries. The compromise of an MCU product may lead to serious consequences such as privacy leakage, financial loss, or even human injury and death. To prevent or mitigate such attacks, extensive security testing during the development phase is imperative.

However, existing firmware security testing approaches demonstrate their own limitations when applied to embedded firmware testing (see Table 1 for an overview of the issues). For example, the emulation-based rehosting technique with application to firmware analysis has been extensively studied in recent years [11, 13, 17, 19, 22, 27, 47], but accurately modeling the behavior of diverse peripherals remains the main research challenge. Several approaches such as P^2IM [17], DICE [27], Laelaps [11], PRETENDER [19], μEmu [47], and Jetset [22] propose to learn an approximate peripheral model using the symbolic execution, access-pattern matching, and machine learning techniques. However, the learned models are inaccurate in general. With inaccurate models, such rehosting approaches cannot boot firmware with complex peripherals such as USB. When the execution trace is not exactly the same as that on the real device, even though the firmware can be “successfully” booted, it is inadequate for many security analysis tasks. The emulation-based rehosting technique is typically used to test *hardware-independent code* after the target firmware has passed through the booting process.

Another direction is to leverage the hardware abstraction layer (HAL) available in MCU firmware to avoid modeling peripherals. For example, HALucinator [13] automatically detects HAL libraries and replaces them with

host implementations. Para-rehosting [43] provides common HAL backend implementations to help port MCU firmware to native hosts. However, both approaches cannot test the peripheral driver which never runs.

Finally, some hardware-in-the-loop (HITL) solutions run the firmware in the QEMU emulator while forwarding peripheral I/O operations to real devices [14, 24, 30, 45]. Although high fidelity is preserved, these HITL-based approaches require frequent and expensive switching (and syncing) between QEMU and hardware, incurring significant performance overhead. For example, the emulation speed is in the order of tens of instructions per second when frequent hardware interaction is needed [45]. As a result, these approaches are typically used to analyze *hardware independent code* after the firmware is fully booted with the help of the forwarding mechanism. To the best of our knowledge, none of the existing HITL work supports firmware fuzzing.

In this work, we propose a new fuzzing solution called *microAFL* (*μAFL*), which is specifically designed for ARM-based MCU devices, to assist developers in locating potential software bugs in the firmware. While our approach can cover the entire software stack, in this paper we focus on its use in testing the low-layer code such as peripheral drivers, which we believe is not properly supported by the existing work. By running the target firmware directly on the target device, our approach supports full-stack testing with high fidelity. *μAFL* requires developers to have access to the prototype development boards and JTAG- or SWD-based debug dongles, which are essential hardware tools available and used in virtually every embedded system development environment.

μAFL is designed to be modularized and extensible so that existing mature fuzzers can be directly integrated (e.g., we adoptAFL[46] in our prototype implementation but any other fuzzers can be used as a drop-in replacement). To achieve this goal, *(i) μAFL* decouples the *execution engine* from the rest of a fuzzer (collectively called *fuzzing manager* in this paper). Specifically, we run the target firmware on the real hardware, and run the fuzzing manager on the PC that coordinates the fuzzing process with the help of the debug dongle. The execution information is also streamed to the fuzzing manager where the analysis is conducted. In this way, we keep the fuzzing manager on the PC agnostic to the execution engine. *(ii)*

To enable communication between the development board and the PC, we re-purpose the debug dongles, which have the highest level of control to the board. Therefore, *μAFL* can efficiently and effectively feed the testcases over the board, pull the run-time execution status, and start/suspend/stop the target. *(iii)* Finally, to collect code coverage information of each testcase, which is essential for grey-box feedback-driven fuzzers, *μAFL* leverages a hardware feature called Embedded Trace Macrocell (ETM) [4] that transparently generates the instruction trace. The trace is streamed directly to the PC via five additional pinouts (four for data and one for clock). Although ETM incurs additional cost, we adopt it in this work for two reasons. First, it enables transparent trace collection. In other words, no code instrumentation is required. This feature makes our work free from rewriting the binary, since most of the third-party libraries are distributed as stripped binaries and no robust binary rewriting tool is available to facilitate instrumentation-based trace collection. Second, we argue that prototype development boards with ETM pinouts are only needed in the development phase. After the firmware has been fully tested, the released products do not need to be equipped with these features. This small investment at the development stage yields a good return on investment (ROI) for manufacturers in the long term, considering the expensive recalls that may happen later.

μAFL features two key components: *online trace collector* and *offline trace analyzer*. They collect ETM data on the device and parse the results on the PC respectively. When a testcase is available on PC (generated by AFL), *μAFL* sends it into a reserved memory on the board via the debug dongle. At the point where the testcase is consumed for the first time, the online trace collector activates ETM to collect the instruction trace, and streams the data to the PC. While collecting the ETM stream, the online trace collector also applies configurable filters via the Data Watchpoint and Trace (DWT) unit [2] to suppress unnecessary ETM packet generation. This not only reduces the amount of tracing data for transmission, but also avoids analyzing useless packets on PC.

The offline trace analyzer runs on the PC and processes the raw ETM data. The result is provided toAFLto maintain the bitmap of code coverage. Decoding the raw ETM data to get the branch information is expensive since it needs to disassemble the firmware and align the instructions with the raw trace [18] (see RQ1 in Section 4). We address this problem by using a kind of special basic block generated at runtime. This allows us to directly use the raw ETM data without disassembling, but still retains path-sensitivity needed to calculate code coverage. The offline trace analyzer also uses a software based approach to filter out uninteresting ETM packets that cannot be filtered by the online trace collector.

Table 1: Comparison with the state-of-the-art solutions

Basic Method	Solutions	Hardware Independent Code	Driver Code	Support Fuzzing	Require Source Code	Require Hardware
Rehosting	HALucinator [13], P^{2}IM [17], μEmu [47], etc.	✓	✗	✓	N	N
Porting	Para-rehosting [43]	✓	✗	✓	Y	N
Forwarding Hardware Interactions	Avatar [45], Avatar^2 [30], SURROGATES [24], Inception [14], etc.	✓	✗*	✗*	N	Y
Fully On-device Execution	*μAFL* (proposed)	✓	✓	✓	N	Y

*: Theoretically, these solutions support fuzzing the driver code. However, the significant overhead on state syncing renders fuzzing driver code impractical. Existing solutions leverage real devices to boot the firmware in QEMU to a state where analyzing hardware-independent code is possible.

We have implemented a prototype of *μAFL* using the SEGGER MCU debugging solution [38]. Then, we used the prototype to test SDKs from two major MCU chip vendors, i.e., NXP Semiconductors [34] and STMicroelectronics [42]. In particular, we used the USB driver fuzzing as a case study in our evaluation. At the time of writing, we have uncovered *13 bugs residing in the USB drivers that were not known previously*. All of them have been confirmed by the vendors and the patches have been released or scheduled with the newest SDK releases.

In summary, our contributions are three-fold:

- We propose *μAFL*, the first fuzzing tool that is applicable to the driver code of MCU firmware. *μAFL* decouples the execution engine from the fuzzing manager so that existing fuzzing tools can be easily integrated.
- We propose using ARM ETM for non-intrusive feedback collection. To improve performance, *μAFL* adopts Linear Code Sequence And Jump (LCSAJ) analysis to directly process raw ETM data without expensive disassembling.
- We have implemented and evaluated our prototype against two SDKs from major MCU chip vendors. We used the USB driver as a case study to show how our prototype can fuzz real-world driver code. The tool has helped us find 13 previously unknown bugs with 8 CVEs allocations.

The source code and the firmware samples used in the evaluation are available at https://github.com/MCUSec/microAFL for future research on this topic.

2. BACKGROUND

2.1 American Fuzzy Lop (AFL)

Fuzz testing is an automated testing technique used to discover coding errors and security vulnerabilities in software. It involves inputting abnormal testcases to the software-under-test in an attempt to make it crash. American Fuzzy Lop (AFL) [28] is one of the most successful fuzzing tools. We roughly split it into two main components for easy presentation: an execution engine and a fuzzing manager. While the former is responsible for running a testcase with the target program, the latter is responsible for generating new testcases by mutation based on a genetic algorithm, coordinating the execution, analyzing the execution information, etc. Concretely,AFLfirst instruments the target program so when the program is executed, the branch information can be generated and recorded. The fuzzing manager then forks a new process as the execution engine to run the program with the current testcase. During execution, the instrumented target program consumes the testcase and records the collected branch coverage information into a local bitmap. The fuzzing manager also aggregates all the local bitmaps into a global bitmap, and compares the newly generated local bitmap to the global one to decide if a new path has been discovered. A testcase that can increase branch coverage is considered interesting and will be used in the genetic algorithm to calculate the subsequent testcases. To report a bug, the fuzzing manager monitors the execution status of the target program and leverages crash information as indicators.

2.2 Analysis of MCU Firmware

MCU is a special-purpose System-on-Chip that cares about realtime processing capability, low power consumption and costs. They are widely used in different application fields, such as wearable, smart home, industrial automation, etc. The execution environment of MCU firmware is significantly different from the traditional OSs, making many existing binary analysis tools includingAFLinapplicable.

Unlike traditional software which assumes an OS layer that provides an abstract view of hardware, MCU firmware runs on bare metal or only includes an OS library (e.g., RTOS) for simple multitask management. Therefore, it compiles the driver code of peripherals and the application code together to form a single-address-space program. The peripheral I/O operation is performed by accessing the memory-mapped registers. Due to the diversity of peripherals, dynamic analysis of MCU firmware is extremely challenging. Although the rehosting technique has made some breakthroughs to test the hardware-independent part of the firmware [11, 13, 17, 19, 22, 27, 43, 47], no existing work can test the driver code.

2.3 Hardware-Supported Instruction Trace Collection

Program instruction trace is helpful in many program analysis applications, such as performance profiling [10, 29], fuzz testing [12], control flow integrity enforcement [18], root cause analysis [15, 16], debugging [32], etc. Compared with software instrumentation, modern processors support capturing the instruction traces by hardware components to reduce the overhead. For instance, Intel incorporates its hardware instruction trace feature, known as Processor Trace or PT [21] to all its Core processors starting from Broadwell. The counterpart of ARM is called *Embedded Trace Macrocell* (ETM) [1] or *Program Trace Macrocell* (PTM) [25][1]. These implementations are quite similar to each other. Both are designed to efficiently rebuild the whole instruction trace assuming that the corresponding machine code is available. More specifically, a dedicated hardware component emits a stream of control flow packets. Then a decoder is used to reconstruct a unique execution path by matching the control flow data to the disassembled machine code.

2.3.1 Trace Collecting. ARM MCUs can optionally implement a buffer for trace storage on the chip, termed *Embedded Trace Buffer* (ETB). However, based on our study, ETB is rarely supported on real chips. Alternatively, ARM also supports streaming the trace data to an external debugger via a physical parallel port, called Cortex Debug+ETM connector [3]. This is the solution used by *μAFL*.

2.3.2 Instruction Flow Reconstruction. To reconstruct the execution flow, a decoder is needed to interpret the trace packets and align them with the disassembled instructions. Control flow packets carry information about a) whether a conditional branch is taken or not, b) the target of an indirect branch, and c) asynchronous events such as exceptions.

Conditional Branches. ARM ETM uses one bit in the *P-header* packet to encode whether the condition of an instruction is true (encoded as `E` or 1) or false (encoded as `N` or 0). True means the corresponding instruction is executed. The reason for this design is that in ARM, almost all instructions can be conditionally executed. Taking the `addeq` instruction as an example, the `add` operation is conducted only if the `Z` flag is set. When this instruction is a branching instruction, such as `beq`, `E` indicates that the branch is taken whereas `N` indicates the branch is not taken.

Indirect Branches. The indirect branch includes indirect calls and function returns. Since the target of an indirect branch can only be determined at run time, ARM ETM emits a packet containing the target address when an indirect branch happens. Such information is encoded into a *branch packet*.

[1] ETM and PTM are similar techniques for different Arm processor lines. We use ETM to refer to both in this paper.

Asynchronous Events. An asynchronous exception could change the control flow at any execution point. Since the current execution location can already be recovered by the *P-headers*, the branch source information is unneeded. In particular, the branch source is calculated by adding the length of executed instructions from the last branch (determined by *P-headers*) to the base address of the last branch target (determined by the previous *branch packet*). ETM encodes asynchronous events using existing branch packets, but extends them with supplementary information. For example, it can indicate whether this branch is caused by an exception rather than a normal call instruction. It also indicates the corresponding exception number if this is an exception. Moreover, ARM MCUs re-purpose existing instructions for exception returns. Put simply, if an instruction results in a control flow transfer of a set of predefined values (EXC_RETURN), then this is treated as a return from exception and the hardware is responsible for fetching the correct target instruction pointer from the exception stack. ETM further emits a *return from exception* packet to encode such an event. With this mechanism, exception entries and returns can be properly paired.

Direct Branches. With the aforementioned information, the decoder can already recover the whole execution flow by aligning the trace data with the disassembled instructions. Note that the trace information about the direct branches is not needed, since the target of a direct branch could be determined by checking the

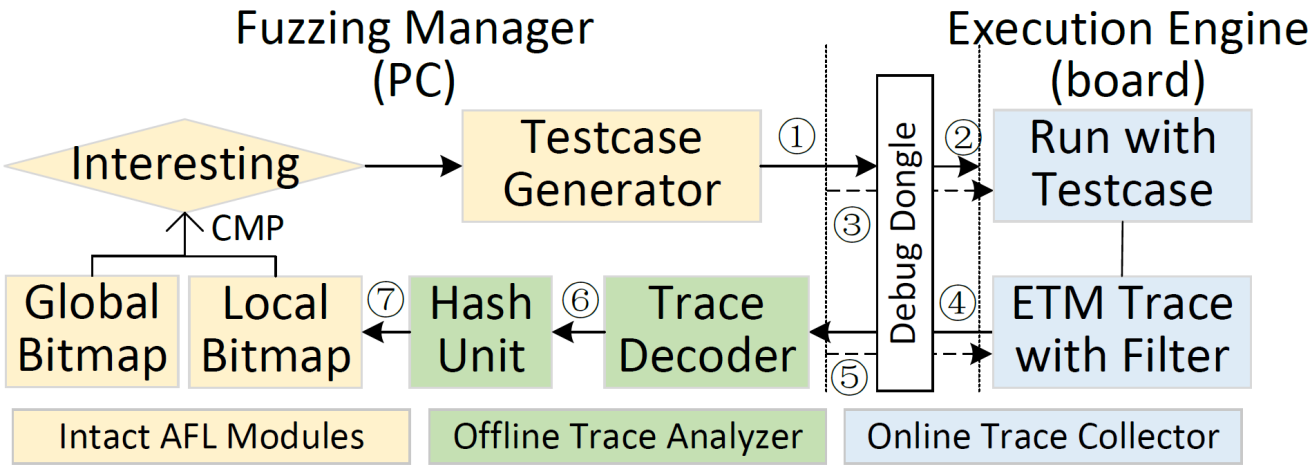

Figure 1: *μAFL* Overview

corresponding branch instruction. However, ETM optionally supports emitting branch packets for direct branches, making it easier to recover the instruction flow at direct branches.

2.3.3 Trace Filtering. It is generally unnecessary to collect the entire instruction trace over time because an analyst might be only interested in a particular code region. Trace filtering allows for suspending trace collection under certain conditions. ARM ETM supports event-based filtering. It defines a set of *ETM event resources* that become active when the corresponding event occurs. These events can be configured by different comparators provided by the hardware. When there is a match, the corresponding event becomes active. For example, when the instruction pointer matches the value in an address comparator, the corresponding event resource is active.

The trace generation is controlled in three ways. First, when an event is active, it can directly enable tracing. When it is inactive, the tracing is disabled. Second, a code region can be included or excluded from tracing. This is achieved by setting a pair of address comparators. Finally, it can be controlled by the *trace start/stop block*. If an event happens,

tracing is started. The tracing does not stop until the block receives a stop signal, which is specified by another event resource. Unfortunately, only the last method is supported by ARM MCUs [1]. Worse, the comparator resources, which are provided by the DWT unit, are very limited. This poses a significant challenge for us to effectively filter the execution trace we care about.

3. μAFL DESIGN

In this section, we begin with an overview of the *μAFL* architecture, and then delve into the detailed design of two critical components, the online trace collector and the offline trace analyzer, as well as their interaction with theAFLframework.

3.1 Overview

μAFL is a new fuzzing tool designed for MCU firmware, with a focus on peripheral driver code. The fundamental idea is to inherit the sophisticated genetic algorithm ofAFLwhile replacing its processbased execution engine with two critical components, an online trace collector and an offline trace analyzer, as shown in Figure 1. This design enables *μAFL* to test all the code in firmware, including peripheral drivers and closed source libraries.

As illustrated in Figure 1, the host PC and target board communicate via a debug dongle, a must-have tool for embedded system development. To begin with, the host PC feeds the testcase via the debug dongle into a reserved memory on the target board (○1) and directs the target to begin execution (○2). Once the target firmware has reached the point where testcase is firstly consumed, the host PC sends the command to activate the ETM function (○3). Then, while the firmware is executing, the generated instruction trace is synchronously streamed to the host PC via the debug dongle (○4). After completing one round of execution, the host PC sends another command to deactivate ETM (○5). The collected trace information is then used to reconstruct the execution paths. *μAFL* adopts a novel scheme for representing branch edges (○6). The final result is mapped into the bitmap to determine whether a new path has been discovered and guide the generation of new testcases following the same genetic algorithm ofAFL(○7).

3.2 Low-level Device Control and Fuzzing Scheduling

We use the JTAG or SWD interface for low-level control of the target device. Through these interfaces, the debug dongle can directly access the processor registers and the device memory (including the memory-mapped system configuration registers) via the Debug Access Port [7]. It also gives us the lowest control over the target device. This is important because fuzzing often causes the target device to enter a non-responsive state. If this happens, we can force a reset via the low-level JTAG/SWD command without human involvement.

We also need to send testcases to the target device. Depending on the generation algorithm, the size of a testcase can be as large as several megabytes. We leverage the SEGGER RTT (real time transfer) protocol [37] for high-speed transmission. Under the hood, RTT uses AHB-AP (Advanced High-performance Bus Access Port) [6] to access memory in the background. Not only can it provide enough bandwidth, but also enables parallel scheduling. Specifically, we transmit the subsequent testcase in the background while the target is running against the current testcase. Moreover, we conduct the analysis of the previous testcase on PC in parallel with the target execution. By scheduling all the tasks in a pipeline, *μAFL* achieves optimized performance.

3.3 Online Trace Collector

The online trace collector is responsible for collecting the ETM instruction trace during the firmware execution. It needs to solve two major challenges: 1) how to feed testcases to the target board from PC, 2) how to selectively collect a minimal but sufficient instruction trace of interesting code snippets.

3.3.1 Testcase Feeding. We reserve two fixed arrays to hold testcases, one for the current test and the other for the subsequent test, which is transmitted in the background during the current test to improve parallelism. The communication channel being used is SEGGER RTT as mentioned before. These arrays are declared in a noinit section so that the libc constructors will not interfere with them during initialization. The size of arrays can be configured as needed.

3.3.2 Trace Collection and Filtering. To collect the execution trace, we can instrument the firmware so that each basic block transition can be recorded and streamed to the PC. However, two challenges need to be addressed. a) The instrumentation requires additional memory space and computing resources that resource-restricted MCU chips may not afford, and b) The current binary rewriting techniques still face some fundamental technical issues (e.g., current disassemblers cannot disambiguate between references and literal values precisely), especially when the target binary is stripped.

To tackle this problem, *μAFL* leverages the ETM hardware feature to generate the instruction trace. The collection is transparent to the firmware. Therefore, no software instrumentation is needed and no additional overhead is incurred. By default, ETM collects all the branch information which is sufficient to recover the full instruction trace of a testcase. However, based on our experiments, this is sub-optimal (see RQ2 in Section 4). In particular, lots of irrelevant packets have to be transmitted and analyzed. For example, the booting process of an MCU is fixed and never influenced by a testcase. We can safely avoid collecting ETM data during device booting to save resources. Moreover, even if we have the abundant resources to do so, the irrelevant packets add noises to the fuzzer that cannot be easily removed. For example, some MCU firmware is multi-tasked. Collecting all the trace information means all the tasks are traced. This brings about non-determinism that leads to a different trace at each run even if the testcase is the same.

We use the DWT hardware feature to filter out irrelevant ETM packets. However, in ARM MCUs, DWT only implements a limited number of comparators (four in a typical implementation) that can be used as filters. We have to prioritize its usage to maximally reduce irrelevant packets. Based on the ETM triggering methods mentioned in

Section 2.3.3, we design two kinds of online filters, namely *address-based filter* and *event-based filter*. When an MCU has more comparator resources, these filters can be combined to generate more fine-grained traces.

Address-based Filter. This filter allows analysts to specify a continuous code region to be traced. It works the best when we are interested in a particular library. This mode consumes two comparators to configure the region start and end.

Event-basedFilter. *μAFL* also supports event-based filters in which certain events trigger the on/off switch of ETM. The event can be either executing an instruction in a particular memory range or reading/writing a particular value from/to a particular address. Both consume two comparators. We call the former instruction trigger and the latter data trigger.

The instruction trigger is very useful in skipping the device booting process. We use the code snippet in Listing 1 as an example. Lines 4-7 are part of device booting and they have nothing to do with the testcase. Line 12 is the main logic of the firmware, which is put in an infinite loop. This is the paradigm in MCU programming – the main operation is executed constantly to sense environmental data and process them accordingly in a loop. We tame the code by adding three lines (9, 13, 14). Note although *μAFL* does not require the source code of the target library, the source code that invokes the target library is needed to make it easy to tame the fuzzing process. fuzz_stop is a flag that marks whether the fuzzing should stop. It can be changed by the firmware when the testcase has been used up or by the debug dongle asynchronously. By configuring the instruction address at line 9 to start ETM and configuring the

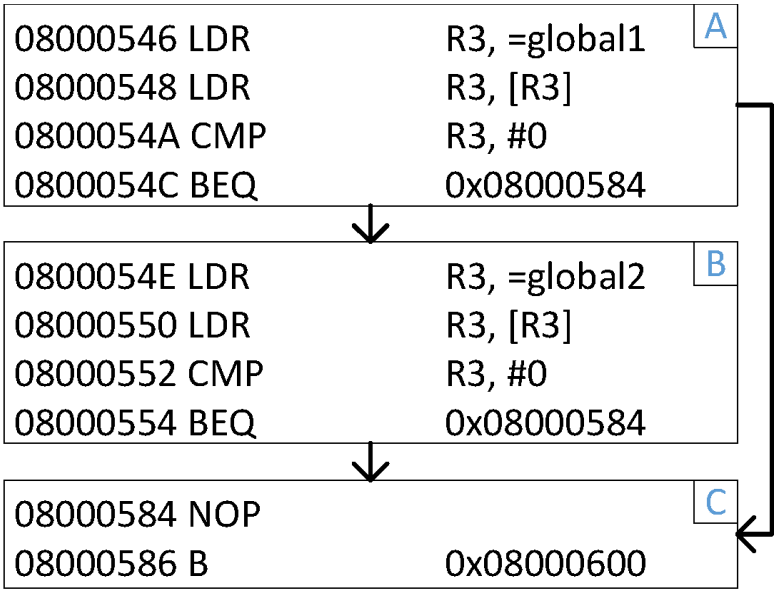

Figure 2: Control flow graph of an example code

instruction address at line 14 to stop ETM, *µAFL* can effectively focus on the main logic of the firmware.

```
17. int fuzz_stop = 0;
18. int main(void)
19. {
20.     MPU_Config();
21.     SCB_EnableICache();
22.     SCB_EnableDCache();
23.     HAL_Init();
24.     // ...
25.     fuzz_stop = 0;
26.     while (1)
27.     {
28.         MX_USB_HOST_Process();
29.         if (fuzz_stop)
30.             break;
31.     }
32. }
```

Listing 1: A code snippet containing initialization code, main application logic, and *µAFL* harness

The data trigger provides fine-grained tracing capability, which we leverage to trace a specified task. More specifically, in the multitask environment, we use the data trigger mode to filter out the execution trace of other tasks and the OS kernel, such as interrupt handlers and scheduling, which is considered as noise to the fuzzer. We observe that in the RTOS environment, each task has its task control block (*TCB*) in a fixed location regardless of the testcase being used. Therefore we can configure DWT to turn on ETM when the global pointer that points to the TCB of the current task (e.g., pxCurrentTCB in FreeRTOS) is written with the TCB address of the target task. When any other value is written to that pointer, indicating the target task is swapped out, ETM is turned off.

3.4 Offline Trace Analyzer

For each testcase execution, the offline trace analyzer processes the ETM packets from the target device. It first recovers the branch information without decoding the raw ETM data. This is achieved using a kind of special basic block. Then, the branch information derived from the basic block transitions is used to update the bitmap of code coverage maintained by AFL. All the otherAFLcomponents are intact, including the new path identification module, the genetic algorithm to generate new testcases, etc.

*3.4.1 Calculating Branch Information without Decoding ETM.AFL*uses the branch (edge) coverage information to identify novel execution paths. To capture branch information,AFL(in QEMU mode) dynamically captures basic block transitions and uses the address of basic blocks to populate the bitmap following the code below [28].

```
1.  cur_location =( block_address > >4) ^( block_address < <8)
2.  ;
3.  shared_mem [ cur_location ^ prev_location ]++;
4.  prev_location = cur_location > >
```

Listing 2: Branch coverage calculation inAFLQEMU mode.

In Listing 2, shared_mem refers to the local bitmap of the current testcase and block_address is the address of the current basic block. In line 1, block_address is fed to a simple hash function to get a random identification of the current basic block, denoted as cur_location. For the path $A \rightarrow B \rightarrow C$ shown in Figure 2, the basic block transition sequence is $0x8000546 \rightarrow 0x800054E \rightarrow 0x8000584$, and for the path $A \rightarrow C$, the basic block transition sequence is $0x8000546 \rightarrow 0x8000584$. Following the code shown above, these two paths generate two different bitmaps, whichAFLcan leverage in finding interesting testcases.

Using ETM trace, we can recover the same branch coverage information. Concretely, by walking along the disassembled instruction sequence and aligning it with decoded ETM packets, we can recover the whole instruction trace and further rebuild the same branch information as AFL. However, this incurs non-trivial overhead according to the literature [18] and it is also verified by our experiments (see Section 4.1).

To avoid expensive code disassembling and ETM decoding, we propose a novel mechanism to capture branch coverage information directly using the raw ETM packets. A straightforward idea is to use the target address of every ETM branch packet as the start of each basic block. However, it cannot differentiate the two paths in Figure 2, because the conditional branches at addresses $0x800054C$ and 08000554 do not generate any branch packet no matter the branch is taken or not. Fortunately, we found in the ETM manual [1] that this behavior can be overridden by setting the eighth bit of the ETMCR register. In particular, with this option enabled, ETM will generate branch packets for direct branches that are actually taken. Therefore, the path $A \rightarrow B \rightarrow C$ would emit a sequence of ETM packets ($0x8000546, EEENEEEE, 0x8000584$),

while the path $A \rightarrow C$ would emit a sequence of ETM packets

($0x8000546, EEEE, 0x8000584$). Here, E is a bit in the P-header which means the condition of an instruction is true while N means the opposite, as mentioned in Section 2.3.2. As can be seen, the latter path directly jumps from the basic block A to C since the branch condition at address $0x800054C$ is true. A branch packet with target $0x8000584$ is thus emitted following the E bit. On the contrary, for the former, the branch condition at address $0x800054C$ is false and therefore no branch packet is generated there. However, the branch condition at address $0x8000554$ is true, which leads to a branch packet with target $0x8000584$. By comparing the two ETM traces, it is obvious that we can differentiate the two paths since the P-header bits in between the two branch targets are different.

To explain the branch coverage information μAFL captures, we first explain a kind of special basic block generated with linear code sequence and jump (LCSAJ) analysis [44]. We call it LCSAJ_BB. A LCSAJ_BB is an instruction sequence starting from the last taken branch target and ending with the following branch instruction which is actually taken. Under this definition, the basic block A alone is a

LCSAJ_BB in the path $A \rightarrow C$, while in path $A \rightarrow B \rightarrow C$, the basic block A concatenated with basic block B constitutes a LCSAJ_BB since the branch at the end of A is not taken. In μAFL, we represent a LCSAJ_BB by combining the base address obtained from the previous branch target and the P-header bitstream before the next

LCSAJ_BB, formally donated as ($BB\_base, BB\_bitstream$). For example, the LCSAJ_BB for A in path $A \rightarrow C$ is encoded as ($0x8000546$, 1111), while the LCSAJ_BB for $A|B$ in path $A \rightarrow B \rightarrow C$ is encoded as ($0x8000546$, 11101111). Note this information can be obtained without referring to the assembly code. In Section 3.4.2, we explain how to use LCSAJ_BB transitions to calculate branch coverage to bridge with AFL. It is worth noting that our approach does not generate the same bitmap asAFLdoes. However, it achieves the same path sensitivity since any change in basic block transitions will be reflected on the change in the corresponding LCSAJ_BB
(either BB_base or $BB\_bitstream$).

Nondeterminism. Although the online trace collector can already filter out a substantial amount of relevant ETM packets, limited by the hardware, it still emits many noisy packets. For example, there is no mechanism to suppress the tracing of exception handlers, which happen non-deterministically. This will add instability to the fuzzer because the same testcase would generate different execution traces in different runs. To address the issue, μAFL also provides an offline exception filter. Specifically, we leverage the exception information embedded in the ETM branch packets to figure out the exception entry points and exit points. The analyst can choose whether or not to discard the ETM trace generated during the handler execution. Again, no disassembling is needed in this process.

3.4.2 Mapping Branch Information to the Bitmap. In this section, we explain how to map LCSAJ_BB-based branch information to the bitmap maintained by AFL. Following theAFLdesign shown in Listing 2, our goal is to transform a LCSAJ_BB denoted by

(BB_base, $BB\_bitstream$) into a unique number, which will be used to replace of role of cur_location (unique identification for the basic block) in Listing 2. To sufficiently diffuse the information contained in each LCSAJ_BB, we adopt a lightweight hash algorithm based on MurMurHash [8]. The output is a random integer BB_ID. As mentioned before, the whole algorithm replaces line 1 of Listing 2. We split the bitstream into chunks of 5-bits and apply bit-wise XOR on them, yielding a number t ranging from 0 to 31

(i.e., FoldAndXor()). Then, t is used to mix with and shift BB_base. The result is further split into two parts and mixed with some magic numbers. While this design is ad-hoc, it effectively randomizes the encoded LCSAJ_BBs such that the resulting BB_ID does not incur too many collisions on theAFLbitmap based on our evaluation.

Algorithm 1: 用于将一个 *LCSAJ_BB* 转换为一个随机整 ID 的哈希函数

Input: $(base, bitstream) \leftarrow$ (*BB_base*, BB_bitstream)
$MAP\_SIZE \leftarrow$ length of bitmap in bytes
Output: *BB_ID*
Function HASH($base, bitstream$):
$t \leftarrow FoldAndXOR(bitstream)$;
$base \leftarrow base + t$;
$left \leftarrow (base \ll (32-t)) \mid (base \gg t)$;
$right \leftarrow (base \ll t) \mid (base \gg (32-t))$;
$BB\_ID \leftarrow (left \mid right)$;
$BB\_ID \leftarrow (BB\_ID \oplus (BB\_ID \gg 16))$;
$BB\_ID \leftarrow (BB\_ID * 0x85ebca6b)$;
$BB\_ID \leftarrow (BB\_ID \oplus (BB\_ID \gg 13))$;
$BB\_ID \leftarrow (BB\_ID * 0xc2b2ae35)$;
$BB\_ID \leftarrow (BB\_ID \oplus (BB\_ID \gg 16))$;
$BB\_ID \leftarrow ((BB\_ID \gg 4) \oplus (BB\_ID \ll 8))$;
$BB\_ID \leftarrow (BB\_ID \wedge (MAP\_SIZE - 1))$;
return *BB_ID*;
End Function

3.5 Crash/Hang Detection

μAFL relies on the built-in exception handling mechanism to detect abnormal firmware behaviors. Specifically, we use the vector catching feature [5] to mark the exceptions in concerns, such as *Hard Fault*, *Mem Manage*, *Bus Fault*, *Usage Fault*, etc. These exceptions indicate critical system errors and thus can be used as crash signals. With vector catching, when such an exception happens, instead of Algorithm 1: Hash function to transform a *LCSAJ_BB* into a random ID

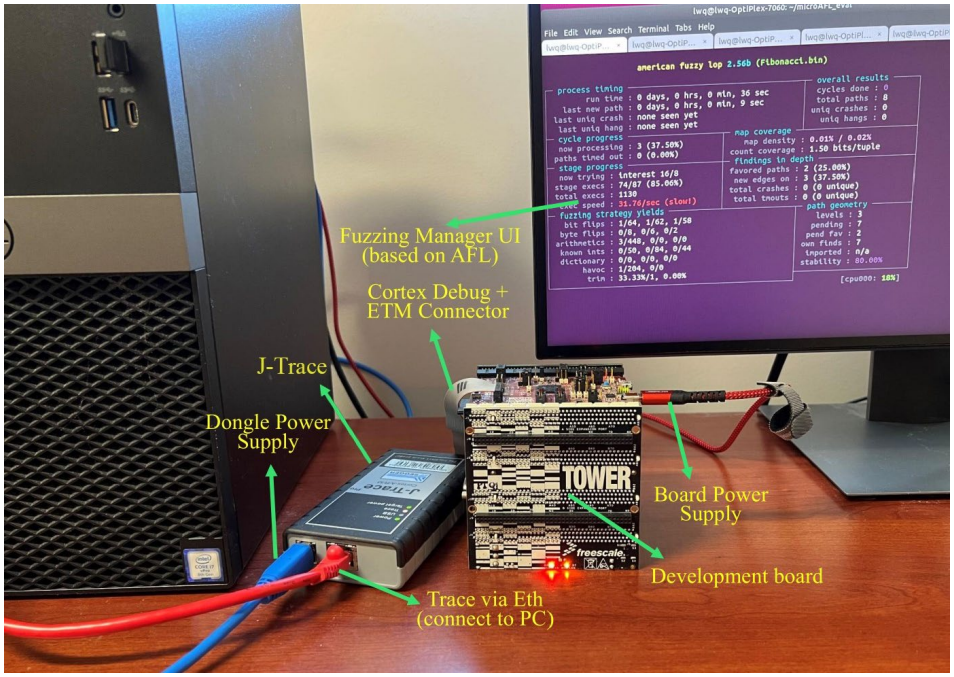

Figure 3: μAFL setup in fuzzing a firmware sample on the NXP TWR-K64F120M board

trapping to the corresponding handlers, the chip enters debug state which can be automatically captured by the debug dongle. Then,

μAFL further checks the Fault Status Registers and Fault Address Registers to examine the root cause. To notify the fuzzing manager of a successful execution, at the end of the tested code, we place a BKPT instruction with a magic number as the argument. If the current test

terminates correctly, the execution will enter debug state at this BKPT instruction. We further check the value of the argument to confirm a successful execution. Lastly, if the fuzzing manager does not capture any debug state in a specified amount of time (we use two seconds as an empirical value), a hang is marked.

4. IMPLEMENTATION AND EVALUATION

We have implemented a prototype of *μAFL* on top of AFL2.56 [46] by adding ~2,000 lines of C code on the PC side. We used the SEGGER J-Trace Pro debug dongle [39] to control the communication between the host PC and the target ARM Cortex-M evaluation boards. The control logic was implemented on the PC side using the SDK provided by SEGGER [38]. In Figure 3, we show the setup in which we used our prototype to fuzz a firmware sample for the NXP TWR-K64F120M evaluation board. The key components are annotated corresponding to the architecture diagram in Figure 1.

We evaluated the performance of our *μAFL* prototype to answer five research questions. The answers are expected to demonstrate the unique strengths of *μAFL* and its effectiveness as a new fuzzing solution for embedded firmware testing.

RQ1: Does the LCSAJ_BB-based approach that directly processes raw ETM trace improve performance compared to approaches that fully recover the instruction flow?

RQ2: Do online filters help to improve the performance of μAFL?

RQ3: How much overhead does μAFL introduce to (each sub-process of) a single round of fuzzing?

RQ4: What is the overall performance of μAFL and how is it compared with existing work?

RQ5: How effective is μAFL in locating bugs in the peripheral drivers of real-world firmware?

Experiment Settings. The PC running the fuzzing manager is equipped with an Intel Core i7-8700 CPU@3.2GHz and 8 GB DDR4 RAM and SSD storage. We used the NXP TWR-K64F120M evaluation board as the execution engine and its corresponding SDKs for the performance experiments from RQ1 to RQ4. To answer RQ5, we present a detailed case study about the real-world bugs that we found after a long-term fuzzing on the evaluation boards NXP TWR-K64F120M and STM32H7B3I-EVAL and towards various projects in their corresponding SDKs. Both boards have ETM pinouts available.

Firmware Samples Used in Evaluation. For RQ1 to RQ4, we used the sample code provided in the NXP SDK. First, the sample Fibonacci implements a recursive function that calculates Fibonacci(1,000,000). This sample does not involve any peripheral and serves as the baseline in our evaluation. Second, the sample I2C involves the usage of a simple peripheral I2C. It merely communicates with the PC over the I2C bus. Third, the sample UART involves the usages of UART, which also communicates with the PC. Forth, the sample USB uses the evaluation board as the USB host to access a USB disk formatted as the FAT file system. Fifth, the sample SD Card recognizes and initializes a micro SD card inserted into the on-board SD card slot. Sixth, the sample Enet uses Ethernet to communicate with the PC over IPv4. Finally, the sample MMCAU uses the hardware Crypto Acceleration Unit (CAU) [33] to complete cryptography operations such as AES, DES3, SHA, etc. Note the library of CAU is closed source. We list the firmware

information including the size and the number of basic blocks in Listing 2. For RQ5, we used the sample code provided in the NXP SDK and STM32 SDK. While we tested multiple drivers in both SDKs, we specifically chose USB as a case study to demonstrate the capability of *μAFL* in finding bugs in complex peripherals. The application-level logic of all the samples is very simple, because we do not attempt to find bugs at high-level code. Rather, we make sure that the core peripheral functions are included in the sample, with the goal to feed abnormal inputs to the low-level driver code to trigger bugs.

Sample	BB#	Size (bytes)	Dis&Dec*	*μAFL* w/o Filter	*μAFL*
Fibonacci	9,927	12,064	100,830	33,386	104,394
I2C	13,343	14,888	112,666	27,732	116,487
UART	9,899	12,856	37,003	21,087	50,571
USB	36,431	40,024	465	2,211	2,236
SD Card	21,104	25,328	592	2,207	2,335
Enet	14,475	18,504	685	1,089	1,116
MMCAU	12,186	17,712	58,155	26,393	74,830

Table 2: * Fully disassemble firmware and decode ETM data. The same filtering mechanism was applied as μAFL.

4.1 Overhead of Fully Recovering Instruction Flow

As discussed in Section 3.4.1, *μAFL* reduces processing overhead by avoiding disassembling the firmware and aligning the ETM trace to recover the full instruction flow. In this section, we demonstrate how using the *LCSAJ_BB*-based approach can reduce performance overhead on PC. In the implementation of the base line approach (fully disassembling firmware and decoding ETM data), we used the popular disassembly framework, Capstone [31]. We followed the firmware execution by aligning the collected ETM packets with disassembled instructions. Whenever a new basic block was met, we disassembled the whole basic block at once, which was also cached for future use. After recovering the instruction trace, we followed Listing 2 to populate the bitmap. It is worth mentioning that we applied the same filtering strategy as *μAFL* to ensure a fair comparison. We measured the total number of executions for each sample in one hour. The results are shown in column 4 of Table 2. Compared with the performance of *μAFL* (column 6), we confirm that *μAFL* is generally much faster than the disassembly-based approach. We observed 1.03x - 4.81x improvement on average.

4.2 Filter Performance

To answer RQ2, we disabled the proposed online filters and compared the performance of *μAFL* with and without filters. As before, we recorded the number of executions within one hour. The results are shown in columns 6 and 5 of Table 2 respectively. First, we can see that online filters improve the performance of *μAFL* in general in all samples that we have tested. For very large samples (e.g., the Enet, USB and SD Card), the filter improves the performance of *μAFL* slightly. However, for less complex samples such as the MMCAU, the improvement becomes very significant. This is because without filters, ETM needs to collect the entire instruction trace beginning from ResetISR to the end of each run, which not only increases the burden of the debug dongle in transferring the raw trace, but also requires more time for the offline decoder to decode and map them into the bitmap. For larger samples, since the execution time is longer for each run, the overhead can be amortized in the entire execution.

4.3 Overhead Breakdown

Each fuzzing round consists of five sub-processes. *(1) Reset2Start* measures the time from device booting (i.e., the start of *Reset_Handler*) to the fuzzing start point (i.e., start reading the testcase). Note we did

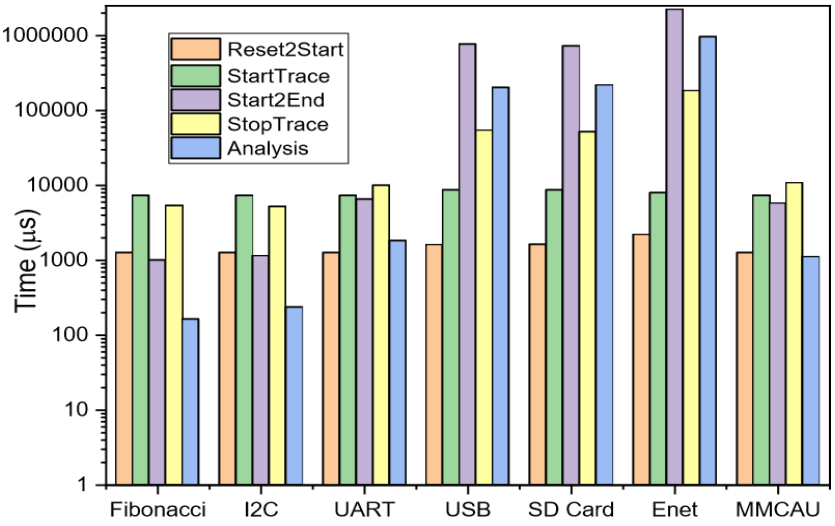

Figure 4: Overhead breakdown of μAFL (y axis is in logarithmic scale).

not collect the branch coverage information during this sub-process since it remains the same among all the testcases. *(2) StartTrace* measures the time spent on preparing a test, including the transmission of the testcase from PC to the board and configuration of ETM and DWT functions. *(3)Start2End* is the time spent from the fuzzing start point to the end point. The branch coverage information is collected for this sub-process. *(4)StopTrace* is the time spent on disabling the ETM trace and waiting for the completion of the ETM trace transmission. *(5) Analysis* is time spent on analyzing the ETM packets.

Project	Time(s)	Executions	Exec/sec	Paths	Crashes/Hangs
I2C	172,893	5,340,552	30.8893	2	0/0
UART	172,838	1,947,980	11.2706	27	1/0
USB	172,803	148,398	0.8588	201	0/500
SD Card	171,693	149,063	0.8682	53	0/0
Enet	174,675	52,961	0.3032	84	14/0
MMCAU	173,180	3,268,671	18.8744	95	0/0

Table 3: Fuzzing performance

In this experiment, we measured the execution time of each subprocess to assess its impact on the overall overhead. We calculated the average execution time over 1,000 rounds of executions for each sample. Figure 4 presents the breakdown of the execution time. We can see that the execution time for the *Reset2Start* and *StartTrace* sub-processes stay stable in all samples, while the execution time for the other sub-processes increases as the firmware becomes more complex. This is because more ETM packets were generated in more sophisticated firmware samples. This will increase the overhead of not only the *StopTrace* operations due to waiting for the completion of the trace transmission, but also the *Analysis* operations due to the increased decoding and hashing computation. We also observed that the execution is very fast for simple firmware. For them, the overhead is dominated by the *StartTrace* and *StopTrace* sub-processes, which are black-box functions provided in SEGGER SDK. As the firmware becomes more complex, it takes longer to complete the *Start2End* sub-process and analyze the massive instruction trace.

Sample	**P^{2}IM**	**μEmu**	**Avatar^2**	***μAFL***
Console	139,860	29,513	1,766	8,261
Fibonacci	209,478	904,617	2,623	8,670

Table 4: Comparison with related work (executions per hour)

4.4 Overall Performance of μAFL

We evaluated the efficiency of the *μAFL* framework in fuzzing realworld firmware samples to answer RQ4. We ran each firmware for around two days and recorded the number of executions, the total execution time, the number of covered paths, and the number of crashes and hangs, as shown in Table 3.

We would also like to point out that, when the generated trace is very large (e.g., over 200 MB), the debug dongle became insufficiently reliable. The buffer inside the debug dongle might be depleted and this led to overflow and trace loss. If this happened, we had to force quit the execution and reset the faulty cycle. This resulted in a lower execution rate since we have to discard the faulty executions. In rare cases, the error in the debug dongle became undetectable. If that happened, we also observed some false positives. This explains the false positives in the table. When we replayed the same testcases that triggered crashes/hangs during fuzzing, the results could not be reproduced.

Comparison with Existing Work. As explained in Section 1,

μAFL is the first work that can efficiently fuzz peripheral drivers for MCU devices. The most related work includes Avatar [45] which emulates the firmware in QEMU but forwards peripheral operations to the real development board, and pure emulation-based solutions such as P^{2}IM [17] and μEmu [47]. The original Avatar only leverages the real hardware to pass the firmware initialization phase and then uses symbolic execution to analyze the code that never accesses peripherals. Later, it was substantially re-engineered for multi-target orchestration purpose in Avatar^2 [30]. However, fuzzing is not supported by default on both. Emulation-based approaches provide great scalability but cannot guarantee sufficient fidelity for fuzzing driver code. For example, firmware with complex peripherals cannot be booted.

In this section, we ignore fuzzing effectiveness, but focus on the raw fuzzing speed achieved by *μAFL*, Avatar^2, P^{2}IM and μEmu. We selected Avatar^2 in our experiments for its active development and more friendly API design. Two samples were used. Apart from the Fibonacci sample mentioned before, we evaluated a Console firmware sample which was also used in P^{2}IM [17] and μEmu [47]. It provides a simple interactive shell via the UART peripheral. It is worth noting that we tried to use the firmware with more complex peripherals such as USB or Ethernet, but none of the related work can support them. Indeed, Avatar^2 does not support DMA which is indispensable for complex peripherals. P^{2}IM and μEmu failed to emulate USB or Ethernet and thus cannot boot the firmware. We augmented the original Avatar^2 framework with fuzzing capability, and also improved the stability of the JLink (the name of SEGGER's debugging solution) target to manage the development board and to coordinate the fuzzing process[2], similar to what we did in *μAFL*. The firmware starts execution in QEMU and forwards the I/O requests to the development board when needed. For P^{2}IM and μEmu, we directly built the systems using the source code indicated in the original papers. However, we made some tweaks in fuzzing Fibonacci. Specifically, P^{2}IM and μEmu mark the end of an execution only when all the bytes

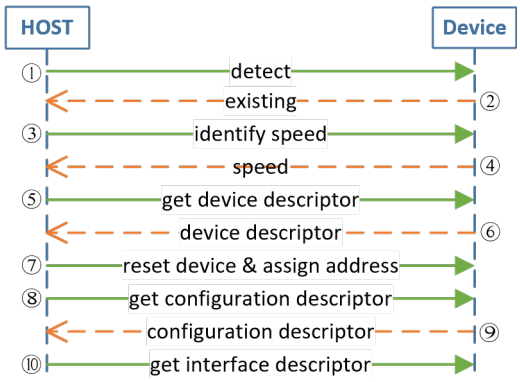

Figure 5: USB Enumeration

[2] https://github.com/avatartwo/avatar2/pull/96

in a testcase are used up. This led to constant time-out in fuzzing Fibonacci. We addressed this issue by manually inserting into the source code many invocations of getchar() to emulate consuming testcases.

The results are summarized in Table 4. Emulation-based solutions outperform hardware-in-the-loop solutions significantly because of the higher computation power. More importantly, they avoid the time-consuming synchronization between the PC and the board. *μAFL* on the other hand outperforms Avatar[2] since it suffers less from synchronization. The gap is narrowed for the computationintensive tasks (Fibonacci) because they do not frequently access peripherals and thus avoid synchronization.

4.5 Real-world Firmware Fuzzing

To demonstrate the capability of *μAFL* in finding bugs in the real world, we fuzzed multiple samples with complex peripheral drivers for two days, including Ethernet, USB and SD card. After running the fuzzer for two days for each sample, we found ten previously unknown bugs in the USB drivers released with the STM32 SDK, and three in NXP SDK. In what follows, we use USB as a case study to show how *μAFL* can efficiently find bugs in complex driver code.

Our fuzzer focused on the USB enumeration process in which the USB device is recognized by the host. In our evaluation, we used the MCU board as a USB host and a SanDisk USB disk as a USB device. We assume the USB disk is malicious and can send arbitrary data to the MCU board. We manually identified the program points where data are received from the USB disk and replaced them with the testcases generated by AFL.

As shown in Figure 5, USB enumeration is very complex and involves multiple rounds of interactions. Most importantly, the USB device is required to send multiple descriptors to specify the device's properties. When the USB host parses these descriptors, memory bugs could occur if the relevant fields are not sanitized properly. In our two-day fuzzing campaign, we have uncovered 13 previously undisclosed vulnerabilities. All of them have been reported to vendors and patched. Note that these bugs influence all the MCU devices using STM32 SDK or NXP SDK. We categorize the bugs we found as follows.

Unsanitized Length Attributes in Descriptors. Lots of bugs were caused by the failure to properly sanitize the length fields in different descriptors. For example, the attribute *bMaxPacketSize* of the device descriptor (○6) defines the allowed maximum size of the packet transmitted from the host to the device. If this value is maliciously specified and not checked, the driver in the MCU host would wrongly configure a faulty pipe and allocate a buffer of unexpected size, causing a denial of service. We found similar bugs in parsing the field *wTotalLength* of the configuration descriptor (○9), interface descriptor (○11) and endpoint descriptor (○13), and the field *wMaxPacketSize* of the endpoint descriptor ○13. The consequences of these bugs range from buffer overflow to

denial of service. The relevant CVEs in this category found by our tool include CVE2021-34259, CVE-2021-34260, CVE-2021-34268, CVE-2021-38258 and CVE-2021-38260.

Missed Hardware Support Checking. The field *bmAttributes* in the configuration descriptor indicates different kinds of power parameters of the configuration. When the respective bit is set, the USB driver attempts to activate the remote wake-up feature in order to save power. However, the driver does not check if the device implements this function, and the system will hang as a result of a failed request. CVE-2021-34261 belongs to this category.

Illogical Endpoint Address. The field *bEndpointAddress* in the endpoint descriptor specifies the identification of the endpoint. The host should generally establish an IN endpoint and an OUT endpoint for data reception and transmission. The driver takes this for granted and does not check the contents of the endpoint descriptor. When a malformed endpoint descriptor specifies the same direction, the host may lose the IN pipe or OUT pipe, leading to system hanging. CVE-2021-34267 belongs to this category.

Unchecked Polling Interval. The filed *bInterval* in the endpoint descriptor indicates the interval for polling endpoint data transmission. It will be used to specify different polling intervals for individual USB applications, such as audio streaming. When this value is negative, the polling logic will hang the device. CVE-202134262 belongs to this category.

5. DISCUSSIONS

Fuzzing Driver Code. Fuzzing peripheral drivers is different from fuzzing normal libraries because peripheral drivers need to interact with the external physical world, which brings lots of nondeterminism in fuzzing. In addition, peripheral behaviors are typically modeled as a state machine, whose state transition is triggered by

many kinds of events such as interrupts and MMIO interactions.

Consequently, effectively fuzzing peripheral drivers needs profound domain knowledge, as we have demonstrated in fuzzing the USB enumeration implementation. Because of this, the code harness for fuzzing a peripheral driver is typically conducted case-by-case. As a general guideline, we should find the program points where the driver interacts with the hardware peripherals. These are places that take inputs from untrusted sources and should thus be tamed to read the testcases. We plan to fuzz more complex drivers in the future.

Applicability. Our approach relies on the ETM hardware feature and therefore cannot be used to test chips without this feature. Fortunately, based on our experience, ETM is very popular among MCU chips. The real problem is that only a small portion of development boards have physical pinouts to interface with ETM, due to the additional cost in PCB design. We argue this does not influence the applicability of our tool in performing in-house testing or fuzzing vendor SDKs. First, developers can easily assemble a PCB board with ETM pinouts [41]. Second, the result of fuzzing an SDK is typically applicable to other SDKs of the same chip vendor. Specifically, chip vendors tend to maintain a common set of device drivers for similar product lines. If a single development board has the ETM pinouts, the discovered bugs could apply to other chips. Taking STM32 MCUs as an example, initially, our tool found CVE-2021-

34268 for the STM32H7 series chips since we only have access to an STM32H7B3I-EVAL board which is ETM-enabled. However, the bug affects almost all the product lines of STM32 chips including STM32F4[3], STM32F1[4], etc.

6. RELATED WORK

6.1 MCU Firmware Fuzzing

Existing approaches for MCU firmware fuzzing can be classified into five categories as shown in Figure 4 of the work by Li et al [43]. Emulation is the most intuitive method. However, full emulation of various MCU in the market is impossible. Existing QEMU-based solutions, such as P[45]IM [17], DICE [27], Laelaps [11], PRETENDER [19], μEmu [47], and Jetset [22] make a trade-off by utilizing a variety of methods, like machine-learning, symbolic execution and so on, to approximate the behavior of the peripherals in order to run the firmware successfully. HALucinator [13] simplifies the process even further by abstracting and replacing the hardware interface without performing any actual peripheral functions. All of them demonstrate good performance and cost-effectiveness. As with HALucinator, para-rehosting [43] also provides a uniform platform by abstracting an MCU on the commodity hardware in order to benefit from the off-the-shelf test suites such asAFL[46] and AddressSanitizer [40] with the most powerful capability. Furthermore, it can greatly improve the fuzzing performance. Both techniques, however, are not able to delve into the driver layer. Although the peripheral forwarding mechanisms (e.g., Avatar [45]) interact with hardware, they are primarily used to support the upper layer's fuzzing. Also, full firmware synchronization between the emulator and the real device will incur additional overhead.

6.2 Peripheral Vulnerability Detection

USBFuzz [36] proposed a framework to apply fuzzing to the commodity OS USB driver by using an emulated USB device in a virtualized kernel. The emulated USB device feeds the testcases and breaks the hardware/software barrier. However, this approach cannot be used in MCU firmware fuzzing, because a) unlike the kernel in a commodity OS host, MCU cannot run any existing fuzzer due to lack of MMU, b) no out-of-box device emulators are available for MCU, and c) MCU peripherals are multifarious and heterogeneous. Therefore, the MCU peripherals still have to be considered individually.

Facedancer [23], an early effort for fuzzing MCU USB drivers, proposed to use a board which responded with a variety of descriptors and disguised itself as different USB devices. Working as the man in the middle between the

[3] https://github.com/STMicroelectronics/STM32CubeF4/blob/

[4] f3b26f16559f7af495727a98253067a31182cfc/Middlewares/ST/STM32_USB_Host_ Library/Core/Src/usbh_ctlreq.c#L355

[5] https://github.com/STMicroelectronics/STM32CubeF1/blob/ f5aaa9b45492d70585ade1dac4d1e33d5531c171/Middlewares/ST/STM32_USB_Host_Library/Core/Src/usbh_ctlreq.c#L360

fuzzer and the target MCU, the board helps to feed the testcase and check the operating state. However, this dummy fuzzing approach is not effective in most cases, since it cannot obtain any feedback from the target to generate the bitmap and guide the testcase generation. Besides, the board supported only USB drivers.

FIRMUSB [20] took a domain-specific approach for detecting USB vulnerabilities. To identify non-compliant behaviors, it contrasted the model generated from known USB databases and retrieved by symbolic execution. So, FIRMUSB could detect vulnerabilities of the binary firmware without running it on the real board. However, similar to FaceDance, this effort focuses on testing USB firmware only.

7. CONCLUSION

We propose *μAFL*, a non-intrusive feedback-driven fuzzing platform for MCU firmware, particularly targeting the low-level peripheral drivers. *μAFL* decouples the execution engine from the originalAFLframework and uses the development board to execute the testcase. To enable communication between the execution engine and the rest of AFL, we leverage the debug dongle commonly available in the embedded system development environment. To effectively obtain instruction trace, we rely on the ETM and DWT hardware features. Finally, we propose using dynamic basic blocks to reduce the overhead of decoding and analyzing the ETM data. We have evaluated our prototype implementation against SDKs from the two popular MCU vendors: NXP and STMicroelectronics. The prototype has helped us find 13 zero-day bugs in the USB stack shipped with the vendor SDKs, and eight CVEs have been allocated.

8. ACKNOWLEDGMENTS

We thank engineers from SEGGER Microcontroller for their tireless technical support in using JLink SDK and permitting us to distribute our work as an open-source project. We thank Dr. Kang Li from Baidu Security and Dr. Kyu Hyung Lee from the University of Georgia for their insightful comments. This work was supported in part by NSF IIS-2014552, the Ripple University Blockchain Research Initiative and a grant from the University of Georgia Research Foundation, Inc.

9. REFERENCES

[1] ARM. 2007. Embedded Trace Macrocell Architecture Specification ETMv1.0 to ETMv3.4. https://developer.arm.com/documentation/ihi0014/latest. (Retrieved: 2022-01-24).
[2] ARM. 2010. Cortex-M4 Technical Reference Manual. https://documentationservice.arm.com/static/5f19da2a20b7cf4bc524d99a. (Retrieved: 2022-01-24).
[3] ARM. 2011. Cortex-M Debug Connectors. https://documentation-service.arm. com/static/5fce6c49e167456a35b36af1. (Retrieved: 2022-01-24).
[4] ARM. 2011. Embedded Trace Macrocell, ETMv1.0 to ETMv3.5. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0014q/IHI0014Q_ etm_architecture_spec.pdf. (Retrieved: 2022-01-24).
[5] ARM. 2020. Fault status registers and fault address registers. https: //developer.arm.com/documentation/ddi0337/e/exceptions/abort-model/faultstatus-registers-and-fault-address-registers. (Retrieved: 2022-01-24).
[6] ARM. 2021. CoreSight Components Technical Reference Manual: AHB-AP. https: //developer.arm.com/documentation/ddi0337/h/debug/about-the-ahb-ap. (Retrieved: 2022-01-24).
[7] ARM. 2021. CoreSight Components Technical Reference Manual: Debug Access Port. https://developer.arm.com/documentation/ddi0314/h/Debug-Access-Port. (Retrieved: 2022-01-24).
[8] Austin Appleby. 2011. MurmurHash. https://sites.google.com/site/murmurhash/. (Retrieved: 2022-01-24).
[9] AZ Defender team. 2021. "BadAlloc" – Memory allocation vulnerabilities could affect wide range of IoT and OT devices in industrial, medical, and enterprise networks. https://msrc-blog.microsoft.com/2021/04/29/badalloc-memoryallocation-vulnerabilities-could-affect-wide-range-of-iot-and-ot-devices-inindustrial-medical-and-enterprise-networks/. (Retrieved: 2022-01-24).

[10] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. Framework for instructionlevel tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*. 154–163.

[11] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. 2020. Device-agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation. In *Proceedings of the 36th Annual Computer Security Applications Conference*.

[12] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. Ptrix: Efficient hardware-assisted fuzzing for cots binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 633–645.

[13] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. 1201–1218.

[14] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD, 309–326.

[15] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA, 17–32.

[16] Yunlan Du, Zhenyu Ning, Jun Xu, Zhilong Wang, Yueh-Hsun Lin, Fengwei Zhang, Xinyu Xing, and Bing Mao. 2020. HART: Hardware-Assisted Kernel Module Tracing on Arm. In *European Symposium on Research in Computer Security*. Springer, 316–337.

[17] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and Hardwareindependent Firmware Testing via Automatic Peripheral Interface Modeling. In *29th USENIX Security Symposium (USENIX Security 20)*. 1237–1254.

[18] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. Griffin: Guarding control flows using intel processor trace. *ACM SIGPLAN Notices* 52, 4 (2017), 585–598.

[19] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. 2019. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *RAID 2019*. 135–150.

[20] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin RB Butler. 2017. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2245–2262.

[21] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/ 64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf. (Retrieved: 2022-01-24).

[22] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. 2021. Jetset: Targeted Firmware Rehosting for Embedded Systems. In *30th USENIX Security Symposium*.

[23] Kate Temkin, Mikaela Szekely. 2020. Facedancer. https://github.com/usb-tools/ Facedancer. (Retrieved: 2022-01-24).

[24] Karl Koscher, Tadayoshi Kohno, and David Molnar. 2015. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. In *9th USENIX Workshop on Offensive Technologies (WOOT'15)*. Washington, D.C.

[25] Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang, and Yunheung Paek. 2017. Using CoreSight PTM to Integrate CRA Monitoring IPs in an ARM-Based SoC. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 22, 3 (2017), 1–25.

[26] MarketWatch, Inc. 2022. IoT Microcontroller (MCU) Market Size 2021 Global Trend, Top Manufacturers, Regions Analysis and Leading 20 Countries and Forecast by 2027. https://www.marketwatch.com/press-release/iot-microcontrollermcu-market-in-2022-113-cagr-with-top-countries-data-what-would-be-thesize-of-iot-microcontroller-mcu-industry-in-2027-latest-126-pages-report2022-01-16. (Retrieved: 2022-01-24).

[27] A. Mera, B. Feng, L. Lu, and E. Kirda. 2021. DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA, 302–318.

[28] Michal Zalewski. 2021. Technical "whitepaper" for afl-fuzz. https://lcamtuf. coredump.cx/afl/technical_details.txt. (Retrieved: 2022-01-24).

[29] Milena Milenkovic, Scott Jones, Frank Levine, and Enio Pineda. 2008. Performance inspector tools with instruction tracing and per-thread/function profiling. In *Linux Symposium*.

[30] Marius Muench, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar^2: A Multi-target Orchestration Platform. In *Workshop on Binary Analysis Research*.

[31] Nguyen Anh Quynh. 2021. Capstone Engine. https://github.com/aquynh/ capstone. (Retrieved: 2022-01-24).

[32] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on ARM. In *26th USENIX Security Symposium (USENIX Security 17)*. 33–49.

[33] NXP Semiconductors. 2021. CAUAP: Crypto Acceleration Unit (CAU) and MmCAU Software Library. https://www.nxp.com/design/developmentboards/tower-development-boards/mcu-and-processor-modules/kinetismodules/crypto-acceleration-unit-cau-and-mmcau-software-library:CAUAP. (Retrieved: 2022-01-24).

[34] NXP Semiconductors. 2021. NXP Semiconductors Official Site. https://www.nxp. com/. (Retrieved: 2022-01-24).

[35] ONE Tech. 2020. WiFi Vulnerabilities on ESP32/ESP8266 IoT Devices. https:// www.onetech.ai/en/blog/wifi-vulnerabilities-on-esp32-esp8266-iot-devices. (Retrieved: 2022-01-24).

[36] Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. 2559–2575.

[37] SEGGER. 2021. J-Link RTT – Real Time Transfer. https://www.segger.com/ products/debug-probes/j-link/technology/about-real-time-transfer/. (Retrieved: 2022-01-24).

[38] SEGGER Microcontroller. 2022. J-Link SDK - Integrate J-Link Support into Applications. https://www.segger.com/products/debug-probes/j-link/technology/jlink-sdk/. (Retrieved: 2022-01-24).

[39] SEGGER Microcontroller. 2022. J-Trace PRO — The leading trace solution. https: //www.segger.com/products/debug-probes/j-trace/. (Retrieved: 2022-01-24).

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 309–318.

[41] SFUPTOWNMAKER. 2022. PCB Basics. https://learn.sparkfun.com/tutorials/pcbbasics/all. (Retrieved: 2022-01-24).

[42] STMicroelectronics. 2021. STMicroelectronics: Home. https://www.st.com/ content/st_com/en.html. (Retrieved: 2022-01-24).

[43] Wenqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi and Fengjun Li. 2021. From Library Portability to Para-rehosting: Natively Executing Open-source Microcontroller OSs on Commodity Hardware. In *NDSS 2021*.

[44] Derek F Yates and Nicos Malevris. 1995. The effort required by LCSAJ testing: an assessment via a new path generation strategy. *Software Quality Journal* 4, 3 (1995), 227–242.

[45] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS'14)*.

[46] Michal Zalewski. 2010. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. (Retrieved: 2022-01-24).

[47] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. In *30th USENIX Security Symposium (USENIX Security 21)*.

[48] ZIMPERIUM. 2018. FreeRTOS TCP/IP Stack Vulnerabilities Put A Wide Range of Devices at Risk of Compromise: From Smart Homes to Critical Infrastructure Systems. https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-put-widerange-devices-risk-compromise-smart-homes-critical-infrastructure-systems/. (Retrieved: 2022-01-24).

[49] ZIMPERIUM. 2018. FreeRTOS TCP/IP Stack Vulnerabilities – The Details. https: //blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/. (Retrieved: 2022-01-24).

毕业论文（设计）文献综述和开题报告考核

导师对开题报告、外文翻译和文献综述的评语及成绩评定：

成绩比例	文献综述 （10%）	开题报告 （15%）	外文翻译 （5%）
分 值			

导师签名_____

年　　月　　日

学院盲审专家对开题报告、外文翻译和文献综述的评语及成绩评定：

成绩比例	文献综述 （10%）	开题报告 （15%）	外文翻译 （5%）
分 值			

开题报告审核负责人（签名/签章）_____

年　　月　　日